

P1031R2: Low level file i/o library

Document #: P1031R2
Date: 2019-06-16
Project: Programming Language C++
Library Evolution Working Group
SG1 Concurrency study group
SG14 Low Latency study group
Reply-to: Niall Douglas
<s_sourceforge@nedprod.com>

A proposal for a low level file i/o library very thinly wrapping kernel syscalls into a portable standard library API, preserving all of the time and space complexities of the host platform. Adopting this proposal into the standard library would substantially improve the i/o performance of standard C++ code (see Introduction), and prevent many race conditions and security violations introduced by the unpredictability of the filesystem. As a low level library, a hosted C++ implementation is not required, and the proposal works well in a [P0829]-type freestanding C++ environment via `embedded_file_source`, which can be used to mark up statically bound file data into an emulated read-only filesystem.

Additional platform features exposed to standard C++ by this proposal include:

1. File locks, including exclusive and shared byte range locking.
2. Memory shared between processes.
3. Virtual memory management and control.
4. Kernel page allocation, permission and page fault control.
5. Large, huge and giant memory pages.
6. Race free filesystem impervious to most third party concurrent modification.
7. Zero whole system memory copy i/o.
8. Device write ordering guarantees to ensure data integrity in the case of sudden power loss.
9. First class support for persistent memory storage.
10. > 10 million item directories, sparse files, allocated file extents management.

The proposal is currently designed around [P0709] *Zero overhead deterministic exceptions* being available, specifically the [P1095] *Zero overhead deterministic failure* proposed implementation thereof. If P0709 is not accepted, it could be easily refactored around `std::expected` instead.

The `mapped<T>` class is dependent on [P1631] *Object detachment and attachment*, which proposes changes to the C++ abstract machine to support mapped storage and shared memory. If P1631 is rejected, that class can be removed from this proposal without damaging the rest of the proposal

(and `map_handle` and `mapped_file_handle` would no longer publicly provide the address of the mapped data). For now, consider the proposal of `mapped<T>` *informative only* until we find out how P1631 fares at SG12.

A reference implementation of the proposed library with reference API documentation can be found at <https://ned14.github.io/llfio/>. It works well on Android, FreeBSD, MacOS, Linux and Microsoft Windows on ARM, AArch64, x64 and x86. Copies of it are shipping in a number of commercial products, where end users have reported very high satisfaction with the design and performance.

Changes since R1:

- Wording for `handle::creation` flags has been improved.
- Incorporated use of proposed `detach_cast<T>`, `attach_cast<T>` and `ensure_stores()` from [P1631] *Object detachment and attachment*.
- Fixed a design flaw in `read()` and `write()` API design whereby million item scatter-gather buffers did not scale well. Thanks to Ben Craig for spotting this.
- Changed timeouts where a partial fill has occurred to no longer return a failure of `errc::timed_out`. This better suits the new non-blocking `io_uring` file i/o API just added to the Linux kernel.
- `mapped<T>` now attaches and detaches the array of `T` in addition to setting up and tearing down the memory map. It also now inherits non-publicly from `span<T>`, the public inheritance of which was a mistake in previous revisions.
- `map_view<T>` has been replaced with `attached<T>` from P1631, which is detailed in that paper instead. `attached<T>` provides a customisation point for third party types to declare how to construct an `attached<T>` from themselves, and `map_handle` and `mapped_file_handle` so declare themselves as sources for `attached<T>`.
- Updated Magnetic vs Flash vs XPoint Storage Capacity per Inflation-adjusted Dollar graph to include 2019 values.
- `safety_fsyncs` renamed to `safety_barriers`.
- `random_file_handle` renamed to `fast_random_file_handle`.

Contents

1	Introduction	5
1.1	Latency to storage has become more important than it was	5
1.2	The immature standard library support for file i/o leads to a lot of inefficient and buggy code and/or reinvention of the wheel	7
2	Examples of use	8
2.1	Read an entire file into a vector assuming a single valid extent:	9
2.2	Write multiple gather buffers to a file:	10
2.3	Map a file into memory and search it for a string (1):	10
2.4	Map a file into memory and search it for a string (2):	11
2.5	Kernel memory allocation and control (1):	12
2.6	Kernel memory allocation and control (2):	13
2.7	Sparsely stored arrays:	14

2.8	Resumable i/o with Coroutines:	15
2.9	Read all valid extents of a file using asynchronous file i/o:	15
3	Impact on the Standard	17
3.1	Non-adopted WG21 proposal dependencies	17
4	Proposed Design	18
4.1	Handles to kernel resources	18
4.1.1	Class hierarchy inheriting from <code>handle</code>	21
4.1.2	Miscellaneous and utility classes and functions	26
4.2	Generic filesystem algorithms and template classes	27
4.2.1	Introduction	27
4.2.2	Filesystem template library (so far) – the ‘FTL’	29
4.2.3	Planned generic filesystem template algorithms yet to be reference implemented	31
4.3	Filesystem functionality deliberately omitted from this proposal	31
5	Design decisions, guidelines and rationale	32
5.1	Race free filesystem	32
5.2	No (direct) support for kernel threads	33
5.3	Asynchronous file i/o is much less important than synchronous file i/o	34
5.4	Pass through the raciness at the low level, abstract it away at the high level	34
6	Draft Technical Specification	34
6.1	Scope	34
6.2	Conformance	35
6.2.1	POSIX conformance	35
6.2.2	Operating system dependent behavior conformance	35
6.3	References	35
6.4	Terms and definitions	36
6.4.1	Cold cache	36
6.4.2	File extents	36
6.4.3	Filesystem entity	36
6.4.4	File serial number	36
6.4.5	Kernel page cache	36
6.4.6	Mapped files	37
6.4.7	Memory page	37
6.4.8	Page fault	37
6.4.9	Storage device	37
6.4.10	File unique id	38
6.4.11	Virtual memory	38
6.4.12	Warm cache	38
6.5	General principles	38
6.5.1	Thinly wrap system calls	38
6.5.2	Zero memory copies	39
6.5.3	Idealised random access storage	39
6.5.4	Genericity in i/o	39

6.5.5	Race free filesystem	40
6.6	Header <io/algorithm/cached_parent_handle_adapter>	40
6.7	Header <io/algorithm/shared_fs_mutex>	41
6.8	Header <io/deadline>	41
6.8.1	Synopsis	41
6.8.2	Class deadline	41
6.9	Header <io/directory_handle>	42
6.10	Header <io/embedded_file_handle>	42
6.11	Header <io/embedded_file_source>	42
6.12	Header <io/file_handle>	42
6.13	Header <io/handle>	42
6.13.1	Synopsis	42
6.13.2	Class handle	46
6.14	Header <io/io_handle>	50
6.14.1	Synopsis	50
6.14.2	Class io_handle	54
6.15	Header <io/map_handle>	64
6.16	Header <io/mapped_file_handle>	64
6.17	Header <io/mapped>	64
6.17.1	Synopsis	64
6.17.2	Class mapped	66
6.18	Header <io/native_handle>	68
6.18.1	Synopsis	68
6.18.2	Class native_handle_type	69
6.19	Header <io/status_code>	70
6.19.1	Synopsis	70
6.19.2	Class file_io_error	70
6.20	Header <io/path_discovery>	70
6.21	Header <io/path_handle>	70
6.22	Header <io/random_file_handle>	71
6.23	Header <io/section_handle>	71
6.24	Header <io/stat>	71
6.25	Header <io/statfs>	71
6.26	Header <io/symlink_handle>	71
7	Frequently asked questions	71
7.1	Why bother with a low level file i/o library when calling the kernel syscalls directly is perfectly fine?	71
7.2	The filesystem has a reputation for being riddled with unpredictable semantics and behaviours. How can it be possible to usefully standardise anything in such a world?	72
7.3	Why do you consider race free filesystem so important as to impact performance for all code by default, when nobody else is making such claims?	73
8	Acknowledgements	73
9	References	73

1 Introduction

Why does the C++ standard need a low level file i/o library, above and beyond needing one to build out an `iostreams v2`?

1.1 Latency to storage has become more important than it was

For a long time now, kernels have kept a cache of recently accessed filesystem data in order to improve read latencies, but also to buffer writes in order to reorder those writes into strides suitable for efficiently making use of a spinning hard drive's actuators. A randomly placed 4Kb i/o to main memory takes up to 5 microseconds, whereas the same i/o to a PMR¹ hard drive takes up to 26,000 microseconds 99% of the time. One could afford a few extra memory copies of an i/o without noticing a difference. Thus the standard library's `iostreams` does not worry too much about the multiple memory copies (in the whole system between the C++ code and the hard drive) that all the major STL implementations make per i/o².

The rise of SSD storage has changed things. Now a SATA connected flash drive takes maybe 800 microseconds for that 4Kb i/o @ 99%³, and random access is as fast as sequential access, so that is no longer an amortised latency figure hiding large individual i/o latency variance. Furthermore, flash based SSDs are highly concurrent, they can service between 16 and 32 concurrent random 4Kb i/o's (queue depth, QD) in almost the same time as a single random 4Kb i/o. These two differences profoundly transform how to write algorithms which work well on a filesystem, but it also has an important consequence for C++:

$$\frac{800 \text{ microseconds}}{32} = 25 \text{ microseconds per 4Kb i/o amortised @ 99\%}.$$

On a SATA connected flash SSD with QD32 i/o, every unnecessary memory copy increases i/o cost by a minimum of 20%!

Achieving sustained QD32 i/o is rare however – one needs to be performing large sequential blocks of i/o of at least $32 \times 4\text{Kb} = 128\text{Kb}$ to have any chance of sustaining QD32, and for large sequential block i/o, latency is usually unimportant for most users⁴.

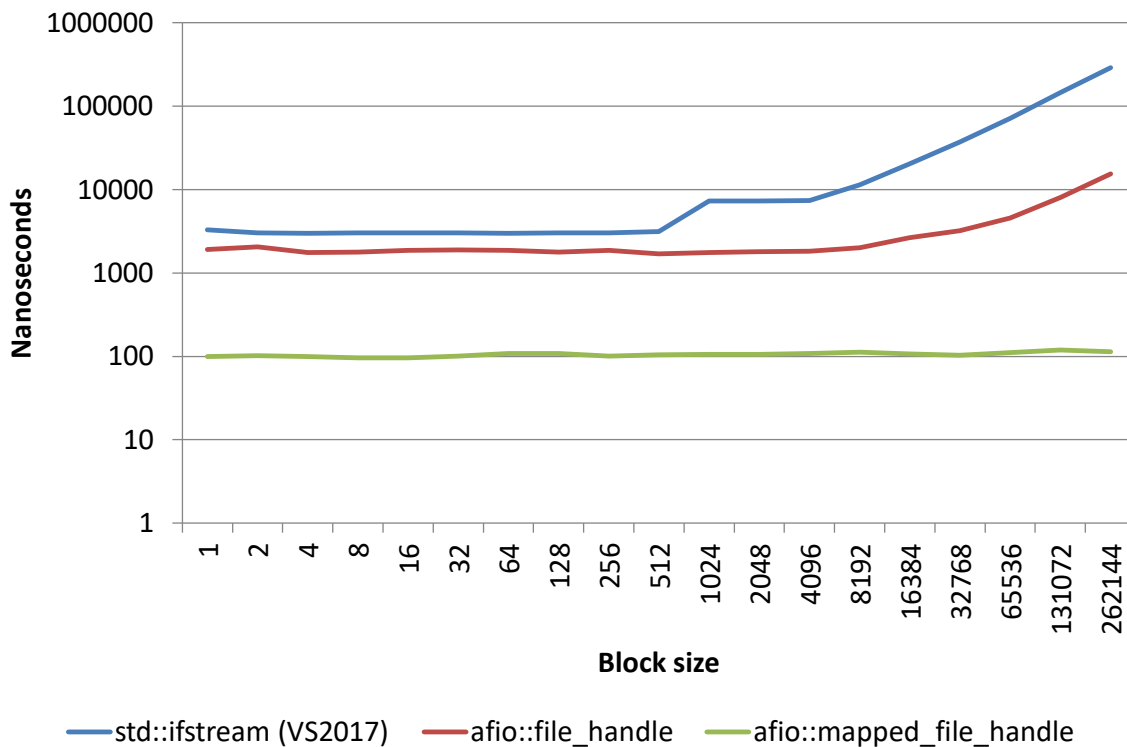
¹Perpendicular magnetic recording. Some of the recent budget large capacity hard drives use Shingled magnetic recording (SMR), these are approximately 15x slower at writes than PMR drives, though they use a 20Gb PMR write cache to hide the drive's true write speed.

²All the major STL implementations implement `std::ofstream::write()` via the C function `fwrite()`. Because of buffering, `fwrite()` often calls `write()` multiple times. Each is an unavoidable memory copy into the kernel page cache, plus kernel transition. Eventually the dirty page in the kernel page cache will reach its age deadline, and be flushed to storage.

³The 99% means that 99% of i/o latencies will be below the given figure. All latency numbers in this section come from empirical testing by me on hardware devices. They differ significantly from manufacturer figures. Device manufacturers tend to quote the latency of the device without intervening filesystem or user space transition. All latency values quoted in this paper include intervening software systems, and are what a user space process can realistically expect to achieve.

⁴But not all. A past consulting client of mine had a problem whereby their application was applying real-time filters to *uncompressed* 8k video at a high frame rate. The CPU demands were not the problem, it was the storage

Figure 1: Latency differential between reads performed using `std::ifstream` and the proposed *Low level file i/o library* as the size of the i/o increases. Test was conducted on a warm cache 100Mb file with random offset i/o, and represents the average of 100,000 iterations. Note the invariance to block size of the low level file i/o library's `file_handle` benchmark up to half the CPU's L1 cache size, demonstrating that no unnecessary memory copies have occurred. Note that the low level file i/o library's `mapped_file_handle` benchmark demonstrates no copying of memory at all.



Block size	1	4	16	64	256	1024	4096	16384	65536
<code>std::ifstream (VS2017)</code>	3301	2986	3017	2994	3020	7254	7389	20282	71538
<code>llfio::file_handle</code>	1915	1766	1869	1873	1855	1750	1812	2633	4576
<code>llfio::mapped_file_handle</code>	99	99	96	108	101	105	108	107	111

However, just recently NVMe rather than SATA connected flash drives have become available to the mass market. These perform that random 4Kb i/o in just 300 microseconds @ 99%. At QD4, which is much more common than QD32, every unnecessary memory copy in the whole system increases i/o cost by 6%. If you are using `<iostream>` on a recent MacBook Pro (which has a high end NVMe flash SSD), perhaps 10% of your i/o cost is due to your choosing `<iostream>`, and especially with larger block sizes it really begins to hurt, as you can see in Figure 1. In my opinion, that is unacceptable in the C++ standard going forward.

And the march of technological progress will make things even worse soon. Intel's NVMe Optane drives using X-Point non-volatile memory will do that 4Kb i/o in just 35 microseconds @ 99%

subsystem: to get smooth video added an unacceptable amount of latency to the real-time video stream for their customers. This is exactly the sort of problem domain C++ ought to excel at.

Magnetic vs Flash vs XPoint Storage Capacity per Inflation-adjusted Dollar 1980-2019

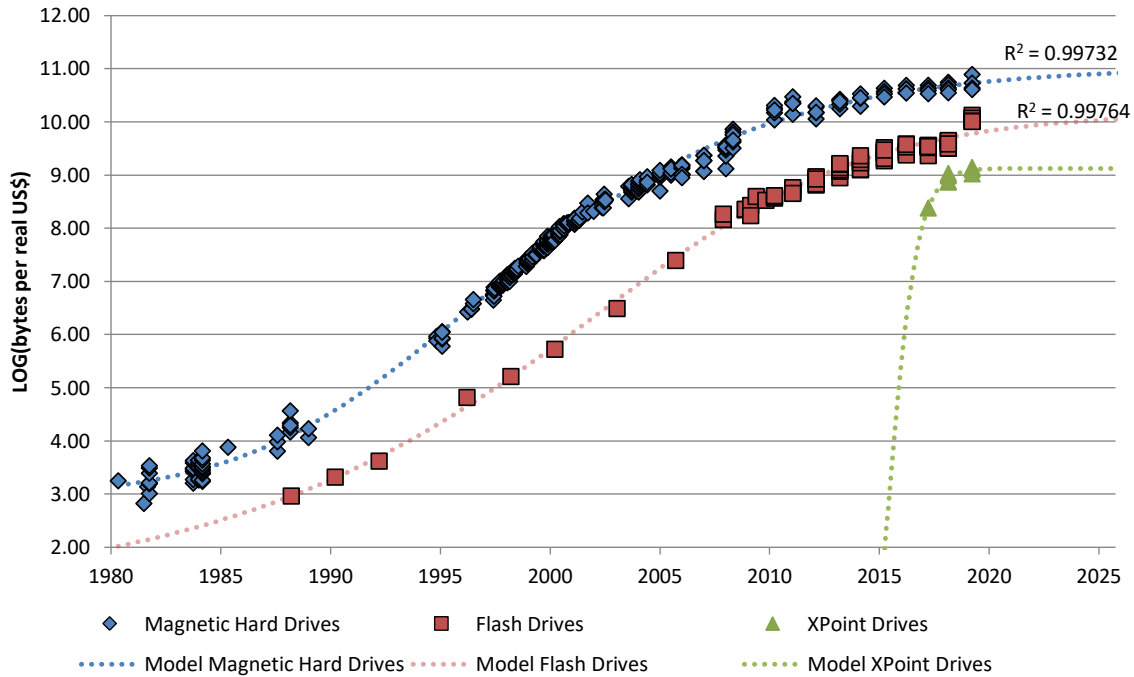


Figure 2: Magnetic vs Flash vs XPoint storage capacity per inflation-adjusted dollar 1980-2019.

and 13 microseconds @ 50%, **and at QD1**. Every unnecessary memory copy in the whole system increases i/o cost by 14-38%.

DDR4 NV-DIMMs have now been standardised, in which your non-volatile storage will do a 4Kb i/o in **8 microseconds**, and indeed a whole 2Mb i/o in just 80 microseconds. Every unnecessary memory copy is now adding **60%** to i/o costs. See Figure 2 for a logistic regression plot of the evolution of storage bytes per inflation adjusted dollar for spinning rust, flash and X-Point technology storage.

If C++ is to achieve the direction laid out in [P0939] *Direction for ISO C++*, in my opinion it needs a data persistence implementation which enables zero memory copies throughout the whole system. One will soon no longer be able to get away with anything less.

1.2 The immature standard library support for file i/o leads to a lot of inefficient and buggy code and/or reinvention of the wheel

Memory mapped files, especially on 64 bit architectures, are usually a good reasonable default choice for most i/o to non-networked drives. They usually have superb sequential and random i/o performance, and usually cause no more than one memory copy in the whole system. Yet using them in C++ – even ignoring the fact that memory mapped files are pure undefined behaviour in

the current C++ standard – is not as trivial as one would imagine. Even with the Boost C++ Libraries to hand, there are two main mechanisms for mapping files into memory, and the plethora of questions about various corner case use issues on Stack Overflow would suggest that neither is entirely obvious to people. They are certainly not ‘fire and forget’, like a `std::ofstream` would be.

One area where a lot of people get stuck is how to efficiently *append* to a memory mapped file. Most developers – probably even most of the WG21 experts reading this paper right now – would suggest making the file much bigger and coordinate between your processes at what offset one ‘appends’ new data. They would suggest this because there is a widespread, *and completely inaccurate*, belief that memory maps are fixed size, and you must tear them down and recreate bigger ones in order to expand a map.

In fact, all the major platforms let you reserve address space for future expansion of a memory map. Indeed, often they will auto-expand your memory map into that reservation if the maximum extent of the backing file is increased, or they provide a super fast syscall for poking the kernel to expand maps of that file across the system. So, as it happens, appending to memory mapped files without costly teardown and recreation of maps is fully supported by kernels, yet judging from Stack Overflow posts, very few realise this⁵.

A standard library supplied implementation of a ‘fire and forget’ memory mapped file primitive object would help address these sorts of problem. The proposed low level file i/o library proposes a suite of polymorphic objects which can perform i/o. Code written to use them need not consider their implementation, thus allowing initiating code to choose whichever implementation is most suitable. Virtual function overrides then choose an optimised implementation, and the code need not worry itself about implementation details. Appends, for example, ‘just work’ with optimal performance for the chosen implementation.

2 Examples of use

A surprising number of people wanted examples of usage before any further discussion of the proposed library design. I therefore supply many such use examples, and my thanks to std-proposals for suggesting which.

I make the following caveats in the following use examples:

- This is a very low level library offering absolute maximum performance, with minimum guarantees of effects, semantics, or behaviours. It is correspondingly less convenient to use. Specifically, no single buffer overloads, no integration with STL containers, no serialisation/deserialisation, no dynamic memory allocation, no (traditional) exception throws. All these convenience APIs, and stronger behaviour guarantees, would be in later standardised layers built on top of this bottom most layer. Please see [P1026] *A call for a Data Persistence (iostream v2) study group* for a broad overview of the vision of which this proposed library is just a foundation.
- There is no file length. Files do not have length. They have a maximum extent *property*. This property refers to the maximum possible extent offset which you will encounter when reading

⁵<https://stackoverflow.com/questions/4460507/appending-to-a-memory-mapped-file>

the valid extents which constitute the file's storage. It is extremely important to understand this difference: files, especially ones built using the planned generic filesystem algorithms template library, may regularly have a maximum extent in the Petabytes range, but store only a few Kb of extents. Algorithms and programs which treat the maximum extent as a length will perform *extremely* poorly in this situation.

This is why we *truncate* files, we do not resize files, because we are truncating those extents exceeding the new maximum extent. We can also truncate to a later maximum extent. I appreciate that many find the idea of 'truncating to extend' confusing, but remember that increasing the maximum extent of a file doesn't actually *do* anything. It simply adjusts a number in the metadata in the inode of the file, and any related kernel resources. It does nothing to the actual file storage. This is why `.extend()` is a poor choice of name, because nothing is extended.

I agree that `.truncate()` is not ideal either, but I feel it is better to focus on the data which could be lost when naming. Better suggestions are, of course, welcome. But do bear in mind that there is a single kernel syscall for changing the maximum extent value, and there is no race free concept of 'set to X if X > Y' etc.

2.1 Read an entire file into a vector assuming a single valid extent:

For brevity, the initial examples are lazy code which will suffer from pathologically poor performance on files with a large maximum extent. Later examples account for allocated extents.

```
1 namespace llfio = std::experimental::io;
2
3 // Open the file for read
4 llfio::file_handle fh = llfio::file(
5     {},          // path_handle to base directory
6     "foo"        // path_view to path fragment relative to base directory
7                 // default mode is read only
8                 // default creation is open existing
9                 // default caching is all
10                // default flags is none
11 );
12
13 // Make a vector sized the current maximum extent of the file
14 std::vector<std::byte> buffer(fh.maximum_extent());
15
16 // Synchronous scatter read from file
17 auto bytesread = read(          // read() found using ADL
18     fh,                        // handle to read from
19     0,                          // offset
20     {{ buffer.data(), buffer.size() }} // Single scatter buffer of the vector
21                                     // default deadline is infinite
22 );
23
24 // In case of racy truncation of file by third party to new length, adjust buffer to
25 // bytes actually read
26 buffer.resize(bytesread);
```

2.2 Write multiple gather buffers to a file:

```
1 namespace llfio = std::experimental::io;
2
3 // Open the file for write, creating if needed, don't cache reads nor writes
4 llfio::file_handle fh = llfio::file(
5     {}, // path_handle to base directory
6     "hello", // path_view to path fragment relative to base directory
7     llfio::file_handle::mode::write, // write access please
8     llfio::file_handle::creation::if_needed, // create new file if needed
9     llfio::file_handle::caching::only_metadata // cache neither reads nor writes of data on this handle
10    // default flags is none
11 );
12
13 // Empty file. Note this is racy, use creation::truncate to be non-racy.
14 fh.truncate(0);
15
16 // Perform gather write
17 const char a[] = "hel";
18 const char b[] = "l";
19 const char c[] = "lo w";
20 const char d[] = "orld";
21
22 fh.write(0, // offset
23     { // gather list, buffers use std::byte
24         { a, sizeof(a) - 1 },
25         { b, sizeof(b) - 1 },
26         { c, sizeof(c) - 1 },
27         { d, sizeof(d) - 1 },
28     }
29     // default deadline is infinite
30 );
31
32 // Explicitly close the file rather than letting the destructor do it
33 // (this throws if it fails, a failure during destruction terminates the process)
34 fh.close();
```

2.3 Map a file into memory and search it for a string (1):

```
1 namespace llfio = std::experimental::io;
2
3 // Open the mapped file for read
4 llfio::mapped_file_handle mh = llfio::mapped_file(
5     {}, // path_handle to base directory
6     "foo" // path_view to path fragment relative to base directory
7     // default mode is read only
8     // default creation is open existing
9     // default caching is all
10    // default flags is none
11 );
12
13 auto length = mh.maximum_extent();
14
15 // mh.address() returns a pointer to indeterminate bytes, so
```

```

16 // we must either bless it, or attach it into the abstract machine
17 span<char> region = in_place_attach<char>({mh.address(), length});
18
19 // Find my text
20 for (char *p = region.data();
21      (p = (char *)memchr(p, 'h', region.data() + length - p));
22      p++)
23 {
24     if (strcmp(p, "hello"))
25     {
26         std::cout << "Happy days!" << std::endl;
27     }
28 }

```

2.4 Map a file into memory and search it for a string (2):

The preceding example used the wrap of other facilities into a convenience type `mapped_file_handle`. For more control and customisation, it can also be done by hand.

Note that we open the same file twice here, one is atomic-append-only, this is used for writes. The other is read-only, and is used to map the file's contents into memory. If you change the file after the map has been established, on some platforms the map needs to be 'poked' to have the map updated to reflect the newly added extents to the end of the file. Portable code will always do this 'poke', it does nothing on most POSIX platforms.

```

1 namespace llfio = std::experimental::io;
2
3 // Open the file for read
4 llfio::file_handle rfh = llfio::file( //
5     {}, // path_handle to base directory
6     "foo" // path_view to path fragment relative to base directory
7         // default mode is read only
8         // default creation is open existing
9         // default caching is all
10        // default flags is none
11 );
12
13 // Open the same file for atomic append
14 llfio::file_handle afh = llfio::file( //
15     {}, // path_handle to base directory
16     "foo", // path_view to path fragment relative to base directory
17     llfio::file_handle::mode::append // open for atomic append
18         // default creation is open existing
19         // default caching is all
20        // default flags is none
21 );
22
23 // Create a section for the file of exactly the current maximum extent of the file
24 llfio::section_handle sh = llfio::section(rfh);
25
26 // Map the end of the file into memory with a 1Mb address reservation
27 llfio::map_handle mh = llfio::map(sh, 1024 * 1024, sh.length() & ~4095);
28

```

```

29 // Append stuff to append only handle
30 llfio::write(afh,
31     0,           // offset is ignored for atomic append only handles
32     {{ "hello", 6 }} // single gather buffer
33                 // default deadline is infinite
34 );
35
36 // Poke map to update itself into its reservation if necessary to match its backing
37 // file, bringing the just appended text into the map. A no-op on many platforms.
38 size_t length = mh.update_map();
39
40 // mh.address() returns a pointer to indeterminate bytes, so
41 // we must either bless it, or attach it into the abstract machine
42 span<char> region = in_place_attach<char>({mh.address(), length});
43
44 // Find my appended text
45 for (char *p = region.data();
46     (p = (char *) memchr(p, 'h', region.data() + length - p));
47     p++)
48 {
49     if (strcmp(p, "hello"))
50     {
51         std::cout << "Happy days!" << std::endl;
52     }
53 }

```

2.5 Kernel memory allocation and control (1):

Something not initially obvious is that this library standardises kernel virtual memory support. This is ‘for free’ as we implement all of the support and control for memory mapped files, and the exact same kernel APIs work with swap file mapped memory (e.g. `mmap()`).

Standardising this support adds lots of interesting opportunities for how STL containers and algorithms which work on reasonably large datasets are implemented.

```

1 namespace llfio = std::experimental::io;
2
3 // Get a kernel page of memory. This may call whatever the equivalent
4 // to mmap() is on this platform to fetch new private memory backed by
5 // the swap file, or it may reuse a page released earlier. The contents
6 // of the returned page is unspecified (a parameter exists to request
7 // that it be all bits zero i.e. always fetch fresh pages from kernel).
8 // Only on first write will a page fault pin a real page for the returned
9 // page.
10 llfio::map_handle mh = llfio::map(4096);
11
12 // Fill the newly allocated memory with 'a' C style. For each first write
13 // to a page, it will be page faulted into a private page by the kernel.
14 // memset() has blessing powers under P0593, so we don't need to do
15 // that by hand.
16 std::byte *p = mh.address();
17 size_t len = mh.length();
18 memset(p, 'a', len);           // blesses the memory into existence

```

```

19
20 // Tell the kernel to throw away the contents of any whole pages
21 // by resetting them to the system all zeros page. These pages
22 // will be faulted into existence on first write.
23 mh.zero_memory({ mh.address(), mh.length() });
24
25 // Do not write these pages to the swap file (flip dirty bit to false)
26 mh.do_not_store({mh.address(), mh.length()});
27
28 // Fill the memory with 'b' C++ style, probably faulting new pages into existence
29 // C++ std::fill() does NOT have blessing powers, so we must
30 // explicitly attach the mapped region before use. This uses the
31 // attached<T> RAII span refinement class from P1631.
32 {
33     p1631::attached<char> p2({mh.address(), mh.length()});
34     std::fill(p2.begin(), p2.end(), 'b');
35     // detaches region on scope exit
36 }
37
38 // Kick the contents of the memory out to the swap file so it is no longer cached in RAM
39 // This also remaps the memory to reserved address space.
40 mh.decommit({mh.address(), mh.length()});
41
42 // Map the swap file stored edition back into memory, it will fault on
43 // first read to do the load back into the kernel page cache.
44 mh.commit({ mh.address(), mh.length() });
45
46 // And rather than wait until first page fault read, tell the system we are going to
47 // use this region soon. Most systems will begin an asynchronous population of the
48 // kernel page cache immediately.
49 llfio::map_handle::buffer_type pf[] = { mh.address(), mh.length() };
50 mh.prefetch(pf);
51
52
53 // You can actually save yourself some time and skip manually creating map handles.
54 // Just construct a mapped<T> directly, this creates an internal map_handle instance
55 // which is automatically attached<T>, so memory is detached and released when
56 // the mapped<T> is destroyed
57 llfio::mapped<float> f(1000); // 1000 floats, allocated used mmap()
58 std::fill(f.begin(), f.end(), 1.23f);

```

2.6 Kernel memory allocation and control (2):

Another thing not initially obvious is that this library standardises shared memory support. This is also ‘for free’ as memory maps are by default shared memory when multiple processes open the same file.

```

1 namespace llfio = std::experimental::io;
2
3 // Create 4Kb of anonymous shared memory. This will persist
4 // until the last handle to it in the system is destructed.
5 // You can fetch a path to it to give to other processes using
6 // sh.current_path()

```

```

7 llfio::section_handle sh = llfio::section(4096);
8
9 {
10 // Map it into memory, and fill it with 'a'
11 llfio::mapped<char> ms1(sh);
12 std::fill(ms1.begin(), ms1.end(), 'a');
13
14 // Destructor unmaps it from memory
15 }
16
17 // Map it into memory again, verify it contains 'a'
18 llfio::mapped<char> ms1(sh);
19 assert(ms1[0] == 'a');
20
21 // It is UB under P1631 "Object detachment and attachment" to have the same
22 // object attached to the abstract machine at more than one address in memory,
23 // so detach the view from the abstract machine. Note that in_place_detach()
24 // is a *customisation point* for which there is a specialisation for
25 // llfio::mapped<T>. Normally throwing away the result of in_place_detach()
26 // is [[nodiscard]] because it is UB to not use the return value, but the
27 // llfio::mapped<T> specialisation of in_place_detach() writes the return
28 // value into the mapped<T> instance, so it is not marked [[nodiscard]].
29 in_place_detach(ms1);
30
31 // Map a *second view* of the same memory
32 llfio::mapped<char> ms2(sh);
33 assert(ms2[0] == 'a');
34
35 // The addresses of the two maps are unique
36 assert(ms1.data() != ms2.data());
37
38 // Yet writes to one map appear in the other map
39 ms2[0] = 'b';
40 // Ensure that the preceding store is not dead store eliminated
41 // using P1631's ensure_stores()
42 p1631::ensure_stores(&ms[0], 1);
43 // Detach ms2, reattach ms1 using P1631
44 in_place_detach(ms2);
45 in_place_attach(ms1);
46 assert(ms1[0] == 'b');

```

2.7 Sparsely stored arrays:

A neat use case making use of the new kernel memory allocation support is for sparsely allocated huge arrays. One can allocate up to 127Tb of address space on most 64 bit architectures.

```

1 namespace llfio = std::experimental::io;
2
3 // Make me a 1 trillion element sparsely allocated integer array!
4 llfio::mapped_file_handle mfh = llfio::mapped_temp_inode();
5
6 // On an extents based filing system, doesn't actually allocate any physical
7 // storage, but does map approximately 4Tb of all bits zero data into memory
8 mfh.truncate(1000000000000ULL * sizeof(int));

```

```

9
10 // Attach a typed view of the one trillion integers
11 // See P1631 for how this works.
12 p1631::attached<int> one_trillion_int_array(mfh);
13
14 // Write and read as you see fit, if you exceed physical RAM it'll be paged out
15 one_trillion_int_array[0] = 5;
16 one_trillion_int_array[999999999999ULL] = 6;

```

2.8 Resumable i/o with Coroutines:

Note that asynchronous file i/o was dropped from this proposal in R1. This use case example remains to remind WG21 members that the reference library can and does implement asynchronous file i/o.

```

1 namespace llfio = std::experimental::io;
2
3 // Create an asynchronous file handle
4 llfio::io_service service;
5 llfio::async_file_handle fh = llfio::async_file(
6     service,
7     {},
8     "testfile.txt",
9     llfio::async_file_handle::mode::write,
10    llfio::async_file_handle::creation::if_needed
11 );
12
13 // Resize it to 1024 bytes
14 truncate(fh, 1024);
15
16 // Begin to asynchronously write "hello world" into the file at offset 0,
17 // suspending execution of this coroutine until completion and then resuming
18 // execution. Requires the Coroutines TS.
19 alignas(4096) char buffer[] = "hello world";
20 co_await co_write(fh, 0, { { buffer, sizeof(buffer) } });

```

2.9 Read all valid extents of a file using asynchronous file i/o:

Note that asynchronous file i/o was dropped from this proposal in R1. This use case example remains to remind WG21 members that the reference library can and does implement asynchronous file i/o.

```

1 namespace llfio = std::experimental::io;
2
3 // Create an i/o service to complete the async file i/o
4 llfio::io_service service;
5
6 // Open the file for read
7 llfio::async_file_handle fh = llfio::async_file( //
8     service, // The i/o service to complete i/o to
9     {}, // path_handle to base directory

```

```

10     "foo"    // path_view to path fragment relative to base directory
11           // default mode is read only
12           // default creation is open existing
13           // default caching is all
14           // default flags is none
15 );
16
17 // Get the valid extents of the file.
18 const std::vector<
19     std::pair<llfio::file_handle::extent_type, llfio::file_handle::extent_type>
20 > valid_extents = fh.extents();
21
22 // Schedule asynchronous reads for every valid extent
23 std::vector<
24     std::pair<std::vector<llfio::byte>, llfio::async_file_handle::io_state_ptr>
25 > buffers(valid_extents.size());
26 for (size_t n = 0; n < valid_extents.size(); n++)
27 {
28     // Set up the scatter buffer
29     buffers[n].first.resize(valid_extents[n].second);
30     for(;;)
31     {
32         llfio::async_file_handle::buffer_type scatter_req{
33             buffers[n].first.data(), buffers[n].first.size()
34         }; // buffer to fill
35         // NOTE: Uses proposed catch() operator from P1095 Zero overhead deterministic failure
36         std::expected ret = catch(llfio::async_read( //
37             fh, // handle to read from
38             { { scatter_req }, valid_extents[n].first }, // The scatter request buffers + offset
39             []( // The completion handler
40                 llfio::async_file_handle *, // The parent handle
41                 llfio::async_file_handle::io_result<llfio::async_file_handle::buffers_type> & // Result of
42                 the i/o
43             ) { /* do nothing */ }
44             // default deadline is infinite
45         ));
46         // Was the operation successful?
47         if (ret)
48         {
49             // Retain the handle to the outstanding i/o
50             buffers[n].second = std::move(ret).value();
51             break;
52         }
53         if (ret.error() == std::errc::resource_unavailable_try_again)
54         {
55             // Many async file i/o implementations have limited total system concurrency
56             std::this_thread::yield();
57             continue;
58         }
59         // Otherwise, throw a file_io_error exception under P0709 Deterministic exceptions
60         throw ret.error();
61     }
62 }
63 // Pump i/o completion until no work remains
64 while (service.run())

```



```

65 {
66     // run() returns per completion handler dispatched if work remains
67     // It blocks until some i/o completes (there is a polling and deadline based overload)
68     // If no work remains, it returns false
69 }
70
71 // Gather the completions of all i/o scheduled for success and errors
72 for (auto &i : buffers)
73 {
74     // Did the read succeed?
75     if (i.second->result.read)
76     {
77         // Then adjust the buffer size to that actually read
78         i.first.resize(i.second->result.read.value().size());
79     }
80     else
81     {
82         // Throw the cause of failure as an exception
83         throw i.second->result.read.error();
84     }
85 }

```

3 Impact on the Standard

Listed at the end of this section are the in-flight WG21 papers this proposal is dependent upon, and which would need to enter the standard before this library can be considered.

3.1 Non-adopted WG21 proposal dependencies

The proposed low level file i/o library has hard dependencies on the following proposal papers not yet adopted into the C++ standard:

1. P0709 *Zero-overhead deterministic exceptions: Throwing values* <https://wg21.link/P0709>.

This proposes that the C++ language implements the lightweight throwing of error/status codes similar to that implemented by Boost.Outcome [1]. The draft Technical Specification wording below assumes P0709 is available.

This dependency *could* be removed, however the API would then become a `std::expected` based design. This is because failure at this low level is common, indeed higher level filesystem algorithms written on top of this library hard require maximum performance failures in order to perform well.

2. P0829 *Freestanding C++* <https://wg21.link/P0829>.

This paper sets out the parts of the C++ language and standard library which are widely compatible with embedded systems. A subset of the proposed low level file i/o library would be available in a freestanding system, specifically `embedded_file_handle` et al which let you create a read-only file system statically linked into the firmware binary.

3. P1028 *SG14 status_code and standard error object for P0709 Zero-overhead deterministic exceptions* <https://wg21.link/P1028>.

This proposes a refactored, even lighter weight `<system_error> v2` which fixes a number of problems which have emerged in the use `<system_error>` as hindsight has emerged. The replacement for `std::error_code`, `status_code`, is rarefied into a proposed `std::error` object for [P0709]. This low level file i/o library in turn uses custom error code domains extending that `std::error` with relevant path and handle information, into a `file_io_error` object which you will see used frequently throughout this paper.

This dependency *could* be removed, perhaps by subclassing `std::error_code`. However, due to exceeding two CPU registers in size, returning such extended error codes would not be as efficient as returning the more compact `status_code`. APIs returning an extended error code would be more awkward to use, and integrate less well with code using them.

4. P1030 *Filesystem path views* <https://wg21.link/P1030>.

This proposes a lightweight view of a filesystem path. Path views can help eliminate the often frequent copying of filesystem paths when calling a library such as this one. This library uses `path_view` almost universally throughout. Removing them would severely penalise proposed `directory_handle::read()`, plus damage the performance and predictability of many of the higher level filesystem algorithms built on top of this library.

5. P1095/N2289 *Zero overhead deterministic failure – A unified mechanism for C and C++* <https://wg21.link/P1095>.

This proposes a specific mechanism for implementing P0709, one based on P1028. Whilst not mandatory for this proposal, our reference implementation library is written around that proposed specific mechanism. If WG21 decides on an alternative implementation of P0709, that should be unproblematic to integrate into this design (the proposed library almost never allocates memory, and so terminate-on-OOM issues don't really apply here).

4 Proposed Design

The low level file i/o library generally works with `span<T>` or `span<span<T>>`, and thus should automatically work well with Ranges.

4.1 Handles to kernel resources

The design is very straightforward and intuitive, if you are familiar with low level i/o. We do not innovate in this proposed design. It is more, or less, a straight thin wrap of a subset of the POSIX file i/o specification, as it was standardised in the POSIX.1-2008 specification (ten years ago was chosen as it has wide implementation conformance), but with significantly weakened behaviour guarantees than those in the POSIX specification. This weakening was done to aid portability, specifically to far-from-POSIX filesystems such as those typically used in HPC and heterogeneous compute.

There is a fundamental type called `native_handle_type` which is a simple, *unmanaged* union storage of one of a POSIX file descriptor, or a Windows `HANDLE`, or some other platform-specific resource identifier type.

`native_handle_type` caches *disposition* about the identifier, specifically what kind it is, what rights it has, is it seekable, does it require aligned i/o, must it be spoken to in overlapped and so on. This is done for performance, as asking the kernel about handle rights is expensive. It can be made invalid i.e. it has a formal invalid state. It is all-constexpr.

At the base of the inheritance hierarchy is the polymorphic class `handle`. It manages a `native_handle_type`, which can be released from its handle if wished. When the handle is destructed, the `native_handle_type` inside the instance is closed.

Class `handle` is a move-only type. It does provide a `clone()` member function which will duplicate the handle. The reason that the C++ copy constructor is disabled is because duplicating handles with the kernel is expensive, and unintentionally doing so would be bad.

Apart from releasing, cloning and closing, the only other thing one can do with a `handle` is to retrieve its current path on the filesystem. It is very important to understand that this is **not** the path it was opened with (if the user wants that, they can cache it themselves). Rather it is what the kernel *says* is the current path for this inode right now⁶. This can be useful to know, as other processes can arbitrarily change the path of large numbers of open files in a single syscall simply by changing the name of a directory further up the hierarchy. In fact, `handle` has entirely trivial storage as it stores nothing which is allocated from memory, it can thus be constexpr constructed.

`Handle` defines many types and bitfields used by its refinements:

- **mode**

This selects what kind of i/o we wish to do with a handle. One of: (i) none (ii) attribute read (iii) attribute read and write (iv) read (v) read and write (vi) atomic append.

- **creation**

This selects what opening a handle ought to do if the path specified already exists or doesn't exist. One of: (i) open existing file entry (ii) only if not exist atomically create and open an entry (iii) create entry if needed but open existing if possible (iv) open existing and atomically truncate to zero, thus retaining creation date and unique filesystem identifier (inode).

- **caching**

This selects what kind of caching (buffering) the kernel ought to perform for this handle:

- No caching whatsoever, and additionally `barrier()` the file and any other related resources⁷ at certain key moments to ensure recovery after sudden power loss (immediately

⁶A standard API for this is not present in POSIX.1-2008, but proprietary APIs are available on all the major platforms and most of the minor ones, including embedded operating systems. For those few systems without kernel support, or where the user explicitly wishes to disable retrieving the current path from the kernel, we provide a templated adapter class `cached_parent_handle_adapter<handle_type>` for all handle types which caches the path for you.

⁷On Linux ext4, one must also sync the parent directory as well as the inode to ensure complete recovery after power loss.

after creation, immediately after maximum extent change, immediately before close).

On many, but not all, platforms this is direct DMA to the device from user space which comes with a list of special use requirements (see later in paper).

- Cache only metadata. On many, but not all, platforms this is direct DMA to the device from user space.
- Cache only reads, and with `barrier()` at key moments described above. Writes block until they and the metadata to retrieve them after power loss fully reach storage.
- Cache reads and metadata, and `barrier()` at key moments described above. Writes block until they fully reach storage, but the metadata to retrieve any newly written extents is written out asynchronously.
- Cache reads, writes, and metadata (the default). Writes are enqueued and written to storage at some later point asynchronously.
- Cache reads, writes, and metadata, and `barrier()` at key moments described above.
- Avoid writing to storage as much as possible. Useful for temporary files.

For those not familiar with data synchronisation outside of `fsync()`, explicitly disabling some or all of kernel caching at handle open results in much better performance than following every write with a `fsync()`. Indeed, in some filing systems like ZFS, a special fast non-volatile device is used to complete an uncached write immediately, which is synced later to slow non-volatile storage.

- **flags**

This selects various bespoke behaviours and semantics:

- `unlink_on_first_close`

Causes the entry in the filesystem to disappear on first close by any process in the system. Microsoft Windows partially implements this in its kernel, and significantly changes how it caches data based on the setting of this flag.
- `disable_safety_barriers`

Disables the safety `barrier()`'s for the modes listed above.
- `disable_safety_unlinks`

Do not compare inode and device with that of the open file descriptor before unlinking it.
- `disable_prefetching`

Most kernels prefetch data into the kernel cache after an i/o. For truly random i/o workloads, this flag ought to be set.
- `maximum_prefetching`

If we are copying a file's contents using caching i/o, this flag ought to be set.

– `win_disable_unlink_emulation`

This flag is present on POSIX, but does nothing.

On very recent editions of Microsoft Windows 10, there is a special kernel call to delete a file with POSIX semantics i.e. its entry is removed from the directory immediately.

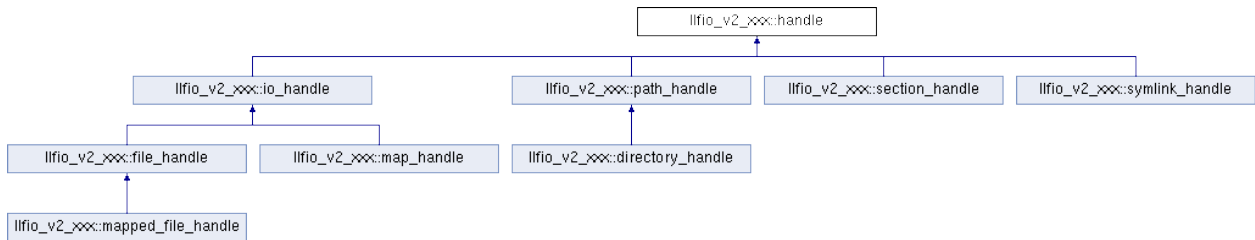
For older editions of Windows, POSIX unlink semantics are emulated by renaming on unlink the file entry to something very random such that it cannot be found⁸. Setting this flag disables this emulation.

– `win_disable_sparse_file_creation`

This flag is present on POSIX, but does nothing.

Microsoft’s NTFS file system was designed in the 1980s back when extents-based filing systems were not common. It was later upgraded to an extents-based implementation capable of working with sparse files. Due to backwards compatibility, during file creation one must *opt-in* to using extents-based storage. That setting remains attached to that file for the remainder of its life, which could theoretically break some programs. The proposed library always opts in to extents based storage by default for newly created files to match semantics with almost every modern filing system elsewhere. This flag disables that default opt-in.

4.1.1 Class hierarchy inheriting from `handle`



Inheriting from class `handle` are these refinements of handle:

- `io_handle`

I/O handle adds types and member functions for scatter-gather synchronous i/o to a seekable handle⁹. All i/o is optionally deadline based, with a choice of interval or absolute timeout.

⁸Due to VMS legacy compatibility, NT implements file deletion by marking a file entry as deleted which prevents it being opened for access thenceforth. It does not remove the file entry until some arbitrary time (usually milliseconds) after the last open handle to it in the system has closed. This confounds code written to expect POSIX semantics whereby unlinking a file causes it to immediately disappear from the filesystem. This workaround of renaming the file to something very random simulates, incompletely, POSIX semantics on Microsoft Windows, sufficiently so at least that most filesystem algorithms ‘just work’.

⁹Non-seekable handles are valid, but that would start to overlap the Networking TS. For various technical reasons, asynchronous socket and pipe i/o cannot portably use the same i/o service implementation as asynchronous file i/o, this is why this proposed library is orthogonal to the Networking TS.

I/O handle also adds member functions for mutually excluding part, or all of, the resource represented by the handle from any other process in the system. These are always *advisory* not mandatory exclusions i.e. they require all processes to cooperate by checking for locks before an i/o.

Inheriting from `io_handle` are these refinements of i/o handle:

– `file_handle`

File handle is the simple, unfussy thin wrap of the platform's file read and write facilities. All i/o is always performed via the appropriate syscall. This passes through any POSIX read-write atomicity and sequential consistency guarantees which may be implemented by the platform.

File handles provide the following additional static member functions:

- * For creating and opening a named file using a `path_handle` instance as the base (a default constructed `path_handle` instance requires the path view to refer to an absolute path).
- * For creating a cryptographically randomly named file at a location specified by a `path_handle` instance. This is useful for creating a temporary file which once fully written to, will be atomically renamed to replace an existing file.
- * For creating a temporary file in one of the temporary file locations found during path discovery (see `path_discovery` below), counted against user quota or system RAM quota.
- * For securely creating an anonymous temporary inode at a location specified by a `path_handle` instance. These are always unnamed, always inaccessible inodes which do not survive process exit. These are used especially by generic template algorithms to implement novel STL containers like vectors with constant, rather than linear, capacity expansion times.

File handles provide the following additional polymorphic member functions:

- * For getting and setting the current maximum file extent (not 'the length', though many people get confused on this).
- * For issuing a write reordering barrier which can be optionally applied to a subset of extents in the file, optionally with blocking until preceding writes reach storage, and optionally with an additional flush of inode metadata which indicates current maximum extent, timestamps etc.
- * For enumerating the valid extents in the file. Modern extents-based filing systems (pretty much all in common use today except for FAT) only store the extents written to, so a 1Tb maximum extent file might only have 4Kb of extents allocated within it. Colloquially known as 'sparse files'.
- * For deallocating a valid extent in the file. Colloquially known as 'hole punching'.
- * For unlinking the hard link currently referred to by the open handle (i.e. deleting).

- * For changing the hard link currently referred to by the open handle to another path, optionally atomically replacing any item currently at that path (i.e. renames).
- * For creating a new hard link to the inode referred to by the open handle at a new path location (i.e. new hard link).

Note that one can instance any refinement of `file_handle` implementation and pass it to functions as if it were a true `file_handle`. Under the bonnet, scatter-gather synchronous i/o is implemented as whatever is the most optimal for that implementation type e.g. for `mapped_file_handle` scatter-gather synchronous i/o is implemented with `memcpy()`.

Inheriting from `file_handle` are these refinements of file handle:

- * `async_file_handle`

~~The async file handle can behave in every way as if a synchronous file handle i.e. the member functions inherited from `io_handle` behave as if synchronous, though unlike in other implementations, they can observe timeouts.~~

~~It adds member functions for scatter-gather asynchronous i/o taking a completion callback (`async_read()`, `async_write()`). Instantiating an async file handle requires the user to supply an instance of `io_service` to issue callback completions against, this must be pumped for completion dispatch very similarly to the `io_service` in the Networking TS.~~

~~Async file handle also provides member functions for coroutinised i/o (`co_read()`, `co_write()`) whereby the calling coroutine is suspended until the i/o completes, whereupon it is resumed.~~

[*Note:* Asynchronous file i/o was removed in R1 of this proposal after feedback from the Rapperswil meeting. – end note]

- * `embedded_file_handle`

The embedded file handle refers to data held in static constant duration storage in the program. It is read-only, but provides an option for it to be opened read-write, whereupon it ignores all writes to the handle instead of failing.

- * `mapped_file_handle`

The mapped file handle is the most highly performing file handle implementation in terms of i/o, but comes with significantly higher cost construction, extension and destruction and with severe usability limits on 32 bit architectures. It also loses any POSIX read-write atomicity and sequential consistency guarantees which may be implemented by the platform on the other types of handle.

It always maps the whole file into memory, extending the map as needed into an *address reservation*. Unless you are opening and closing files frequently, or the files you are working with are much smaller than the system page size, or you are on a 32 bit architecture, this is an excellent default choice for most users giving maximum zero whole system memory copy performance on all devices apart from network attached storage devices.

- * `fast_random_file_handle`

The fast random file handle synthesises read-only file data by hashing a random or user-supplied nonce with the offset requested in the i/o. This creates the appearance of a file full of random data. It is read-only, but provides an option for it to be opened read-write, whereupon it ignores all writes to the handle instead of failing.

The main use case for this handle is for unit test suites, where one often needs an ‘infinite’ sized file, or a maximum possible performance source of file data, to test code for correctness. The ‘fast’ refers to the random data not being cryptographically random, but as a result it approaches main memory in maximum transfer bandwidth on sufficiently endowed CPUs (e.g. AVX512).

- `map_handle`

Map handle is a region of shared or private memory mapped from a backing `section_handle`, or unmapped private memory backed by the swap file, or reserved address space. Within the committed (i.e. allocated) part of that region, i/o can be performed, or more usefully, the region can be accessed directly as memory.

Added member functions include the ability to commit (allocate) sub-regions of reserved address space, or to decommit (deallocate) previously allocated sub-regions.

Map handle can map small, large, huge, massive and super pages to the same extent as the host operating system kernel.

It comes with a comprehensive set of static member functions which can be applied to any memory in a process e.g. ‘please kick the contents of this memory page out to backing storage’, ‘please unset the dirty bit of this memory page (i.e. don’t flush its contents to storage until the next modification)’, or ‘please asynchronously ready this range of memory for access (i.e. prefault it)’ and so on.

`mapped_file_handle` and many other classes use this class as an internal implementation primitive for all forms of mapped and unmapped and reserved memory.

- `path_handle`

Path handles refer to some base location on the filesystem from which path lookup begins. The inode opened may change its path arbitrarily and at any time without affecting the paths which use an open path handle as their base. This handle is, therefore, the foundation of the race free filesystem which the proposed library implements.

Many platforms implement the creation of these handles as an especially lightweight operation, hence they are standalone from `directory_handle`.

Inheriting from `path_handle` are these refinements of path handle:

- `directory_handle`

Directory handles refer to inodes which list other inodes. The main added member function is to read that list of other inodes into a user supplied array (`span`) of `directory_entry`. One can open existing directories, create new directories, create randomly named new

directories, and in your choice of path including temporary paths found during path discovery. One can of course also unlink and relink directories.

- **section_handle**

Section handles refer to a section of shared or private memory. They may be backed by a user supplied `file_handle`, or by an anonymous inode in one of the path categories returned by `path_discovery`, or by some other source of shared memory. They are particularly useful for when you need some temporary storage (counted against either the RAM quota or the current user's quota) which will be thrown away at process end.

Section handles have a length which can be queried and changed. It may be less than, but cannot exceed, the maximum extent of any backing file.

Section handles have additional flags in addition to those inherited from `handle`. Section handle flags are reused by `map_handle`:

- **none**: This memory region is reserved address space.
- **read**: This memory region can be read.
- **write**: This memory region can be written.
- **cow**: This memory region is copy-on-write (i.e. when you first write, the kernel makes you a process-local copy of the page).
- **execute**: This memory region can contain code which the CPU will execute.
- **nocommit**: Don't immediately allocate resources for this section/memory region upon construction. Most kernels allocate space for unbacked sections against the system memory + swap files, and will refuse new allocations once some limit is reached. Setting this flag causes unbacked sections to allocate system resources 'as you go' i.e. as you explicitly commit pages using the appropriate member functions of `map_handle`.
- **prefault**: Prefault, as if by reading every page, any views of memory upon creation. This eliminates first-page-access latencies where on first access, the page is faulted into existence.
- **executable**: This section represents an executable binary.
- **singleton**: A single instance of this section is to be shared by all processes using the same backing file. This means that when one process changes the section's length, all other processes are instantly updated (with appropriate updates of maps of the section) at the same time, which can be considerably more efficient.
- **nvr**: Assume that this section represents non-volatile memory, and use i/o semantics appropriate for that type of memory. Setting this flag provides much superior performance on persistent memory based hardware, plus it activates various kernel options where appropriate to provide sudden power loss safety on persistent memory.
- **page_sizes_1**: Use `utils::page_sizes()[1]` sized pages, or fail.
- **page_sizes_2**: Use `utils::page_sizes()[2]` sized pages, or fail.

- `page_sizes_3`: Use `utils::page_sizes()[3]` sized pages, or fail.
- `barrier_on_close`: Maps of this section, if writable, issue a blocking `barrier()` when destructed, blocking until data (not metadata) reaches physical storage.

- `symlink_handle`

Symlink handles refer to inodes which contain a relative or absolute path. Added member functions can read and write that stored path.

4.1.2 Miscellaneous and utility classes and functions

There are also some utility classes:

- `deadline`

A deadline is a standard layout and trivially copyable type which specifies either an interval or absolute deadline. Deadlines can construct from any arbitrary `std::chrono::duration<>` or `std::chrono::time_point<>`. The advantage to this object is halving the number of polymorphic function overloads required, and maintaining a stable ABI.

- `directory_entry`

A `path_view` and `stat_t` combination. Filled by `directory_handle`'s `read()` function. Note that it has standard layout and is trivially copyable.

- `page_sizes()`

A function returning const lvalue ref to a `vector<size_t>` representing the page sizes currently available to the calling process on this machine.

- `path_discovery`

Path discovery generally runs once per process and it interrogates the platform to discover suitable paths for (i) storage backed temporary files (counted against the current user's quota) and (ii) memory backed temporary files (counted against available RAM). Path discovery does not trust the platform specific APIs, and it tries creating a file in each of the directories reported by the platform to find out which are valid. This is slow, so the results are statically cached.

It is exposed directly so performance sensitive code can call it during a cold code path, in order to prevent library functions which need to find temporary file folders etc. from performing discovery during hot code paths.

- `path_view`

Path views are covered in detail in [P1030], but in essence they are a lightweight reference to a string which is the format of a filesystem path. They are standard layout and trivially copyable. Path views are very considerably more efficient to work with than filesystem path objects, and make a big difference to performance, especially when enumerating large directories.

- `stat_t`

Almost certainly WG21 will want the name to be changed to avoid conflict with the platform `stat_t`, but I haven't personally found it to be an issue in practice. This is a C++-ified `struct stat_t`, it uses `std::filesystem` constants and data types instead of the platform-specific ones. It is standard layout and trivially copyable.

One has the ability to stamp an open handle with parts of a `stat_t`, as well as fill parts of a `stat_t` from an open handle.

- `statfs_t`

Similarly, almost certainly WG21 will want the name to be changed to avoid conflict with the platform `statfs_t`, but I haven't personally found it to be an issue in practice. This is a C++-ified `struct statfs_t`, it uses `std::filesystem` constants and data types instead of the platform-specific ones. Unusually for types in the proposed library, this one is not trivially copyable as it contains two `std::string`'s and a `std::filesystem::path` for the `f_fstypename`, `f_mntfromname` and `f_mntonname` members.

There are some minor utility functions as well which are not described in detail for now. They have the kernel return single TLB entry allocations of varying sizes either via a C malloc type API or via a special STL allocator, ask the kernel to fill a buffer with cryptographically strong random data, fast to-hex and from-hex routines and so on. These minor utility functions are used throughout the internal implementation of the library, but are useful to other code built on top of the library as well.

4.2 Generic filesystem algorithms and template classes

4.2.1 Introduction

A key thing to understand about this low level library is the lack of guaranteed behaviours it provides in its very lowest layers. This is principally because file i/o has surprisingly few guarantees in the POSIX standard, and thus we are gated as to what the thin kernel syscall wraps can guarantee. For example, `file_handle::barrier()` asks the kernel to issue a write reordering barrier on a range of bytes in the open file, with options for blocking until preceding writes reach storage, and whether to also flush the metadata with which to retrieve the region after sudden power loss. This looks great, but you will find wide variation as to how well that is implemented across platforms. These are the current behaviours on the three major platforms¹⁰:

- FreeBSD/MacOS

For normal files, range barriers are not available, so the whole file is barriered. Metadata is always synchronised. On MacOS only, non-blocking barriers are available, on FreeBSD all barriers always block until completion of the entire file plus metadata. On FreeBSD a total sequentially consistent ordering is maintained, so concurrent barriers exclude other barriers until completion. I do not know the behaviour on MacOS, but I would assume it is the same.

¹⁰This is from memory, it may be inaccurate.

For mapped files, range barriers are only available if not synchronising metadata, in which case it is to the nearest 4Kb page level. Blocking until writes reach storage forms a sequentially consistent ordering, otherwise concurrent barriers are racy.

- Linux

For normal and mapped files, fully implemented to the nearest 4Kb page level. BUT with the huge caveat that these do not form a total sequential ordering amongst concurrent callers upon overlapping byte ranges, so it is therefore racy in terms of useful recovery after sudden power loss.

It is common on Linux to silently ignore barriers for code running inside virtual machines on publicly shared hypervisors, especially LXC containers as they are a source of denial of service attack. Filing systems on Linux may also be mounted with hardware barriers disabled, in this situation the storage device is not required to observe the ordering of writes issued to it by the kernel.

In other words, on Linux `barrier()` may *appear* to work, but in fact silently does **not** work, and has no effect.

- Microsoft Windows

For normal files, range barriers are not available, so the whole file is barriered. Otherwise full implementation, and a total sequentially consistent ordering is maintained so concurrent barriers exclude other barriers until completion.

For mapped files, range barriers are only available if not synchronising metadata, in which case it is to the nearest 4Kb page level. Concurrent barriers are always racy.

Filing systems on Windows may be mounted with hardware barriers disabled, in this situation the storage device is not required to observe the ordering of writes issued to it by the kernel.

What this means is that on Linux or if barriering on a mapped file, you must coordinate between multiple processes or threads using your own mechanism to ensure only one thing issues a barrier for some range at a time. On all platforms apart from Linux, currently range barriers with metadata actually barrier the whole file, so there is no point in trying to achieve any concurrency in your write reordering barriers.

In case you think this sort of platform specific variance is limited to just write reordering barriers, you may be in for a surprise. In my own personal opinion (explained in more detail below), I don't think any standards text can specify what a function must do, but only what it must **not** do, for all the lowest level functions. Even the humble write data function has a multitude of platform specific surprise (see proposed standardese for `write()` below).

These variations may seem problematic, but it is exactly what generic filesystem algorithms and template classes are for: to add layers of increasing abstraction plus stronger guarantees on top of the raw low level API. That way, for those who need the raw bare metal performance, they can get that. But for more portable code where we need some consistency, template algorithms can abstract out these platform specific details for us.

As an analogy, in the Networking TS we have lowest level functions such as `async_write_some()`

which attempts to write some or all of a gather buffer sequence. But we also have higher level functions – `async_write()` – which guarantees to write a whole gather buffer sequence, not completing until it is all done. That design pattern of API layers of increasing guarantees is present in file i/o as well, just a bit more complex than (and quite different to) socket i/o.

4.2.2 Filesystem template library (so far) – the ‘FTL’

These are some generic algorithms and template classes which act as abstraction primitives for more complex filesystem algorithms. It should be stressed that all of the below are 100% header only code, and use **no** platform-specific APIs. They are implemented **exclusively** using the public APIs in the proposed low level file i/o library. This may give an idea of the expressive power to build useful and interesting filesystem algorithms using the proposed design.

- `shared_fs_mutex`

This is an abstract base class for a family of shared filing system mutexes i.e. a suite of algorithms for excluding other processes and threads from execution using the filesystem as the interprocess communication mechanism.

Unlike memory-based mutexes already in the standard library, in the lock operation these mutexes take a sequence of *entities* upon which to take a shared or exclusive lock. An entity is a 63 bit number (the top bit stores whether it is exclusive or not)¹¹.

The reason that these mutexes are list-of-entities based is because it is very common to lock more than one thing concurrently on the filing system, whereas with memory-based mutexes that is the exception rather than the norm. For example, if you were updating file number 2 and file number 10 in a list of files at the same time, you would concurrently lock entities 2 and 10. If you were implementing a content addressable database like a git store, you’d use the last 63 bits of the git SHA as the entity, and so on.

Each of the implementations has varying benefits and tradeoffs, including the ability to lock many entities in the same time as one entity. The appropriate choice depends on use case, and to an extent, the platform upon which the code is running.

- `shared_fs_mutex::atomic_append`

This implementation uses an atomically appended shared file as the IPC mechanism. Advantages include invariance to number of entities locked at a time, ability to sleep the CPU and compatibility with all forms of storage except NFS. Disadvantages include an intolerance to one of the using processes experiencing sudden process exit during lock hold, and filling all available free space on filing systems which are not extents based (i.e. incapable of ‘hole punching’).

- `shared_fs_mutex::byte_ranges`

This implementation uses the byte range locks feature of your platform as the IPC mechanism. Advantages include ability to sleep the CPU and automatic handling of sudden

¹¹This design choice works around the problem that on some platforms, byte range locks are *signed* values, and attempting to take a lock on a top bit set extent will thus always fail.

process using during lock hold. Disadvantages include wildly differing performance and scalability between platforms, lack of thread compatibility with POSIX implementations other than recent Linux, ability to crash NFS in the kernel due to overload.

– `shared_fs_mutex::lock_files`

This implementation uses exclusively created lock files as the IPC mechanism. Advantages include simplicity and wide compatibility without corner case quirks on some platforms. Disadvantages include an inability to sleep the CPU, and an intolerance to one of the using processes experiencing sudden process exit during lock hold.

– `shared_fs_mutex::memory_map`

This implementation uses a shared memory region as the IPC mechanism. Advantages include blazing performance to the extent of making your mouse pointer stutter. Disadvantages include inability to use networked storage, inability to sleep the CPU, and an intolerance to one of the using processes experiencing sudden process exit during lock hold.

– `shared_fs_mutex::safe_byte_ranges`

This implementation – on POSIX only – wraps the byte range locks on the platform with a thread locking layer such that individual threads do not overwrite the locks of other threads within the same process, as is required by the POSIX standard for byte range locks. On other platforms, this is a typedef to `shared_fs_mutex::byte_ranges`.

• `cached_parent_handle_adapter<T>`

Ordinarily, handles do not store any reference to their parent inode. They provide a member function which will obtain a such a handle by fetching the current path of the inode and looping the check to see if it has a leaf with the same inode and device number as the handle. This, obviously enough, is expensive to call.

For use cases where a lot of race free sibling and parent operations occur, one can instantiate any of the handle types using this adapter. It overrides some of the virtual functions to use a cached parent inode implementation instead. These parent inode handles are kept in a global registry, and are reference counted to minimise duplication. This very considerably improves the performance of race free sibling and parent operations, at the cost of increasing the use of file descriptors, plus synchronising all threads on accessing the global registry.

There is an additional use case, and that is where the platform does not implement file inode path discovery reliably, which can afflict some older editions of some kernels ¹².

• `mapped<T>`

A `mapped<T>` is an owning view of a `map_handle`'s region, having attached an array of type `T` from the underlying bytes upon construction using [P1631]'s `in_place_attach()`. Upon

¹²At the time of writing, OS X's path fetching API returns one of the paths for any hard link to the inode, randomly. This is almost certainly a bug. FreeBSD does not reliably provide path fetching for file inodes, but does for directory inodes. From examination of the kernel source, this ought to be easy to fix. In both cases, fetching the path of a directory inode is reliable, and thus via this adapter works around these platform-specific quirks and bugs.

destruction, it ensures stores have completed to the region via P1631's `ensure_stores()`, detaches the array of T via P1631's `in_place_detach()`, and destroys the embedded `map_handle` to tear down the mapped region. This is basically a convenience combination of P1631's `attached<T>` with a `map_handle`.

The same rules from [P1631] therefore apply for attaching the same bytes more than once. It is undefined behaviour to simultaneously construct more than one `mapped<T>` or `attached<T>` referring to the same underlying bytes.

4.2.3 Planned generic filesystem template algorithms yet to be reference implemented

- Persistent page allocator which is interruption safe, concurrency safe, lock free. This is effectively a persistent linked-list implementation of allocated and non-allocated regions within the file.
- The B+ tree implementation mentioned in the Introduction [2] which is interruption safe, concurrency safe, lock free.
- Persistent vector which is interruption safe, concurrency safe, lock free.
- Coroutine generators for valid, or all, file extents.
- Compare two directory enumerations for differences (Ranges based).
- B+-tree friendly¹³ directory hierarchy deletion algorithm.
- B+-tree friendly directory hierarchy copy algorithm.
- B+-tree friendly directory hierarchy update (two and three way) algorithm.

4.3 Filesystem functionality deliberately omitted from this proposal

The eagle eyed will have spotted entire tracts of the filesystem have been omitted from this initial proposal:

- Permissions

Standardising this is a ton of extra work best pushed, in my opinion, into a later standardisation effort.

- Extended attributes

These probably could be standardised without much effort, but I am also unsure of the demand from the user base. Despite almost universal support in file systems nowadays, they are not widely used outside of MacOS, which is a shame.

¹³By 'B+-tree friendly', I mean that the algorithm orders its operations to avoid the filesystem's B+-tree rebalancing frequently, as a naïve algorithm which almost everybody writes without thinking will do. This can improve performance by around 20% on the major filing systems.

- Directory change monitoring

This is surprisingly hard to implement correctly. Imagine writing an implementation which scales up to 10M item directories and never misrepresents a change? The demands on handling race conditions correctly are very detailed and tricky to get right in a performant and portable way. I would like the change delta algorithms decided upon before tackling this one.

5 Design decisions, guidelines and rationale

The design decisions are as follows, in priority:

5.1 Race free filesystem

As anyone familiar with programming the filesystem is aware, it is riddled with race conditions because most code is designed assuming that the filesystem will not be changed by third parties during a sequence of operations. Yet, not only can the filesystem permute at any time, it is also a bountiful source of unintended data loss and security exploits via Time-of-check-Time-of-use (TOCTOU) failures.

As an example, imagine the following sequence of code which creates an anonymous inode to temporarily hold data which will be thrown away on the close of the file descriptor, perhaps to pass to a child process or something:

```
1 int fd = ::open("/home/ned/db/foo", O_RDWR|O_CREAT|O_EXCL, S_IWUSR);
2 ::unlink("/home/ned/db/foo");
3 ::write(fd, child_data, ...);
```

Imagine that privileged code is executing that code. Now witness this:

```
1 int fd = ::open("/home/ned/db/foo",
2     O_RDWR|O_CREAT|O_EXCL, S_IWUSR);
3
4
5 ::unlink("/home/ned/db/foo"); // oh dear!
```

```
1
2
3 ::rename("/home/ned/db", "/home/ned/db.prev");
4 ::symlink("/etc", "/home/ned/db");
```

We have just seen unintended data loss where `/etc/foo` is unlinked instead of the programmer intended `/home/ned/db/foo`.

Here is another common race on the filesystem:

```
1 int storefd = ::open("/home/ned/db/store",
2     O_RDWR);
3
4
5 int indexfd = ::open("/home/ned/db/index",
6     O_RDWR);
```

```
1
2
3 ::rename("/home/ned/db", "/home/ned/db.prev");
4 ::rename("/home/ned/db.other", "/home/ned/db")
5 ;
```


Now the index opened is not the correct index file for the store file. Misoperation and potential data corruption is likely.

POSIX.1-2008, and every major operating system currently in use, fixes this via a *race free* filesystem API. Here are safe implementations:

```
1 int dirh = ::open("/home/ned/db", O_RDONLY|O_DIRECTORY);
2 int fd = ::openat(dirh, "foo", O_RDWR|O_CREAT|O_EXCL, S_IWUSR);
3 ::unlinkat(dirh, "foo", 0);
```

```
1 int dirh = ::open("/home/ned/db", O_RDONLY|O_DIRECTORY);
2 int storefd = ::openat(dirh, "store", O_RDWR);
3 int indexfd = ::openat(dirh, "index", O_RDWR);
```

The proposed low level file i/o library considers race free filesystem to be sufficiently important that it is enabled by default i.e. it is always on unless you explicitly ask for it to be off. The natural question will be ‘How expensive is this design choice?’.

These are figures for the reference library implementation running on various operating systems and filing systems. They were performed with a fully warm cache i.e. entirely from kernel memory without accessing the device. They therefore represent a **worst case** overhead.

	FreeBSD ZFS	Linux ext4	Win10 NTFS
Delete File:	6.2%	11.6%	0%

The extra cost on POSIX for deletion is due to opening the inode’s parent directory, checking that a leaf item with the same name as the file to be unlinked has the same inode and device as that of the open handle, and if so then unlinking the leaf in that directory. This algorithm makes file deletion impervious to concurrent third party changes in the path, up to the containing directory, during the deletion operation. A similar algorithm is used for renames, and added overhead is typically around 10%.

One will surely note that overhead on Microsoft Windows is zero. The is because the NT kernel provides much more extensive a race free filesystem API than POSIX does. In particular, it provides a by-open-file-handle API for deletion and renaming so one need not implement any additional work to achieve race freedom.

I appreciate that the choice to make race free filesystem opt-out rather than opt-in will be a controversial one on the committee, not least due to implementation concerns on the less major kernels¹⁴. However it is my belief that correctness trumps performance for the default case, and for those users who want the fastest possible filesystem performance, race free filesystem can be disabled per object in the constructor.

5.2 No (direct) support for kernel threads

[*Note:* Asynchronous file i/o support was removed after feedback from Rapperswil, and thus this section no longer applies. It will be removed in the next revision. – end note]

¹⁴See the description of `cached_parent_handle_adapter<T>` above. However I believe that kernel maintainers are highly amenable to adding a syscall to unlink-by-fd or relink-by-fd, they just need to be given a business case for it. It certainly is trivially easy to implement in any of the kernel sources I have investigated.

5.3 Asynchronous file i/o is much less important than synchronous file i/o

[*Note*: Asynchronous file i/o support was removed after feedback from Rapperswil, and thus this section no longer applies. It will be removed in the next revision. – end note]

5.4 Pass through the raciness at the low level, abstract it away at the high level

Anyone with experience with the file system knows how racy many of the kernel syscalls are. For example, enumerating valid extents on POSIX is utterly racy due to a particularly bad choice of enumeration API design. There are races in anything which involves a filesystem path, by definition, but there are also races in the ordering of reads and writes to a file, the reported maximum extent of a file, and lots more races in what order all changes land on non-volatile storage, which affects recoverability after sudden power loss.

It is not the business of a low level library to hide this stuff. So pass it through, unmodified, and supply higher level layers, templates, and algorithms which abstract away these core problems.

6 Draft Technical Specification

A highly incomplete, work in progress, draft TS wording follows for *Low level file i/o*. The following in-progress WG21 papers – or near equivalents thereof – are assumed to have been standardised in the presented wording:

1. [P0709] *Zero-overhead deterministic exceptions: Throwing values*
2. [P1028] *SG14 status_code and standard error object for P0709 Zero-overhead deterministic exceptions*
3. [P1030] *Filesystem path views*
4. [P1095] *Zero overhead deterministic failure – A unified mechanism for C and C++*

Like in many of my other WG21 papers, we also assume the presence of a new language feature, **bitfield**. This is the combination of an enumeration with individual bits in an unsigned integral value. It is a ‘stand-in’ for some future standardised bitfield in the language.

6.1 Scope

[llfio.scope]

This Technical Specification specifies requirements for implementations of an interface that computer programs written in the C++ programming language may use to perform operations on file systems and their components, such as paths, regular files, and directories. This Technical Specification is applicable to information technology systems that can access hierarchical file systems, such as those with operating systems that conform to the POSIX ([llfio.norm.ref]) interface. This Technical Specification is applicable only to vendors who wish to provide the interface it describes.

6.2 Conformance

[llfio.conformance]

Conformance is specified in terms of behavior. Ideal behavior is not always implementable, so the conformance sub-clauses take that into account.

6.2.1 POSIX conformance

[llfio.conformance.posix]

Some behavior is specified by reference to POSIX ([llfio.norm.ref]). How such behavior is actually implemented is unspecified.

[*Note:* This constitutes an ‘as if’ rule allowing implementations to call native operating system or other API’s. – end note]

Implementations are encouraged to provide such behavior as it is defined by POSIX. Implementations shall document any behavior that differs from the behavior defined by POSIX. Implementations that do not support exact POSIX behavior are encouraged to provide behavior as close to POSIX behavior as is reasonable given the limitations of the operating systems and file systems available to the vendor. If an implementation cannot provide any reasonable behavior for more than a small subset of this specification, an implementation of this library ought to not be provided by the vendor.

6.2.2 Operating system dependent behavior conformance

[llfio.conform.os]

Some behavior is specified as being operating system dependent ([llfio.def.osdep]). The operating system an implementation is dependent upon is implementation defined. It is permissible for an implementation to be dependent upon an operating system emulator rather than the actual underlying operating system.

6.3 References

[llfio.references]

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

- ISO/IEC 14882, *Programming Language C++*
- ISO/IEC 9945, *Information Technology – Portable Operating System Interface (POSIX)*

[*Note:* The programming language and library described in ISO/IEC 14882 is herein called the C++ Standard. The operating system interface described in ISO/IEC 9945 is herein called POSIX. – end note]

This Technical Specification mentions commercially available operating systems for purposes of exposition.¹⁵

¹⁵POSIX® is a registered trademark of the IEEE. Mac OS® is a registered trademark of Apple Inc. Windows® is a registered trademark of Microsoft Corporation. This information is given for the convenience of users of this

Unless otherwise specified, the whole of the C++ Standard’s Library introduction [lib.library] is included into this Technical Specification by reference.

6.4 Terms and definitions [llfio.terms]

For the purposes of this document, the terms and definitions given in the C++ Standard and the following apply.

6.4.1 Cold cache [llfio.terms.cold_cache]

This refers to the situation where any of the contents and/or metadata relating to a filesystem entity have not been cached into kernel memory, and an operation relating to that entity would require the kernel to communicate with the storage device (which may take a non-deterministic period of time).

6.4.2 File extents [llfio.terms.extents]

A file’s ([fs.def.file]) contents are stored as a sequence of zero to many *allocated extents* (some older filing systems can only use zero or one allocated extent). An allocated extent is a region bound in between offset zero and the maximum extent property of the file (often incorrectly called ‘file length’) which may store non-zero bits. The regions between allocated extents are not stored on the device, and appear to i/o as all bits zero.

6.4.3 Filesystem entity [llfio.terms.entity]

This refers to any single collection of extents on the filesystem which has a POSIX file serial number, and an entity includes any metadata specifically describing it held by other entities. This includes files ([fs.def.file]) and directories ([fs.def.directory]). Note that entities may have zero, or many, canonical paths ([fs.def.link]) on the filesystem.

6.4.4 File serial number [llfio.terms.inode]

The POSIX file serial number is an unsigned integer type identifying an item of storage on a filesystem. It is defined by POSIX to be unique to the specific filesystem it is within (but not unique within the whole system). A colloquial term for it is ‘inode’.

6.4.5 Kernel page cache [llfio.terms.page_cache]

Operating system kernels may cache the data read from, and written to, a filesystem entity. This may be a single coherent unified cache, or multiple incoherent caches. Metadata may be cached while

document and does not constitute an endorsement by ISO or IEC of these products.

extents are not cached. The reference to ‘page’ is that these caches typically work in granularities of a memory page, so if extents are cached, it is to the nearest memory page alignment and the cached region will be some round multiple of the memory page.

On unified page cache architecture kernels, mapping a file into memory directly maps a portion of the kernel’s own page cache for that file’s extents into the process. When a C++ program reads or writes a mapped file, it directly works with the kernel’s cache of that file.

6.4.6 Mapped files [llfio.terms.mapped_files]

POSIX provides facilities to map the contents of some or all of a file’s extents into the address space of a C++ program. Reads from the region read the data in the file; writes to the region modify the data in the file.

6.4.7 Memory page [llfio.terms.memory_page]

Many CPUs will map physical memory to the virtual addresses seen by the C++ program in units of a memory page. A CPU may support one, or many different sizes of memory page.

[*Note:* For information, the memory page sizes for a x64 processor are 4Kb, 2Mb and 1Gb, with a potential additional future memory page size of 512Gb. For an ARM Cortex A9 CPU, the memory page sizes are 4Kb, 64Kb, 1Mb and 16Mb. Other ARM CPUs vary. For RISC-V, 4Kb + 4Mb or 4Kb + 2Mb + 16Gb memory page sizes are the most common. – end note]

6.4.8 Page fault [llfio.terms.page_fault]

Virtual memory systems implement work-on-demand via *page faulting*. Each memory page in a C++ process can be marked by an operating system kernel to call the kernel upon the first read, or write, inside a memory page. The kernel may then perform a number of actions. A very common action to handle a page fault is allocating memory for that page and making the new allocation available at that address, before resuming execution. Another very common action is that for writable memory maps, to mark a page as dirty and needing later flushing to physical storage.

Page faults significantly complicate the estimation of upper bounds of execution times by introducing a stochastic factor. In some cases, regions of memory can be *prefaulted* in order to eliminate the potential for unexpected page faults later on.

6.4.9 Storage device [llfio.terms.storage]

This is the hardware which retains the state of the filesystem across power loss events. It may, on some systems, be the same as random access memory.

6.4.10 File unique id

[llfio.terms.unique_id]

Under POSIX, each file has an unsigned unique identification number which is guaranteed to be unique anywhere on the currently running system. POSIX defines it to be a combination of the device id, and the file serial number.

6.4.11 Virtual memory

[llfio.terms.virtual_memory]

This is the simulation of there being far more memory available to a C++ program than there is in reality. It typically works in granularities of a memory page, and POSIX provides quite a few control functions to enable C++ programs to manipulate virtual memory directly. Portable wrappers of many of these control functions are provided by this specification.

6.4.12 Warm cache

[llfio.terms.warm_cache]

This refers to the situation where **all** of the contents and metadata relating to a filesystem entity have been cached into kernel memory, and no further read operations relating to that entity would be performed to the storage device. In this situation, any non-modifying operation ought to have bounded execution times, as described in [llfio.principles.latency_preserving].

6.5 General principles

[llfio.principles]

This is a low level library intended mainly as a building block for higher level C++ standard library constructs such as Ranges, Containers, and Serialisation. It can be used directly by C++ programs, but at the cost of having to do more work by hand than if the program used the higher level construct instead.

6.5.1 Thinly wrap system calls

[llfio.principles.latency_preserving]

This specification is designed to wrap proprietary kernel system calls into a set of *common C++ functions* which are portable across those platforms upon which C++ is implemented. User space overhead between calling the functions in this specification, and calling the kernel system calls directly, shall be statistically unmeasurable in real world use cases, except where documented otherwise.

Specifically:

1. The user space overhead to handle a kernel system call failure shall also be statistically unmeasurable in real world use cases.
2. The functions in this specification shall be *latency preserving*, unless documented to not be so. Latency preservation means that the first 99.99% of the shape of the sorted distribution of execution latencies for repeated calls of the underlying kernel system call shall be preserved

by the functions in this specification. A constant vertical shift upwards to reflect constant time processing etc is, however, permitted.

Where a function is documented to be latency degrading, the potential causes of this latency degradation shall be documented by this specification.

The most noticeable consequence of this principle is that we do not allocate memory, except where documented. Where memory is required for an operation, we accept buffers supplied by the user instead. This means that some functions may be latency degrading in their default parameter set, but would be latency preserving if called with the right non-default parameters.

6.5.2 Zero memory copies

[llfio.principles.zero_copy]

This specification avoids causing the copying of any form of memory wherever possible, even where it is inconvenient to the library user. We do not copy filesystem paths, scatter-gather buffer lists, nor data input/output, except where absolutely unavoidable.

6.5.3 Idealised random access storage

[llfio.principles.idealised]

This specification models an idealised random access storage device, one whose curve of sorted distribution of read latencies exactly matches that of the computer's random access memory. The only difference over reads from random access memory is a constant added vertical shift.

[*Note:* It is important to note that writes are not considered in this idealisation. They have an implicit memory allocation, and thus are not easily predictable. – end note]

Empirically, reads from a warm cached file (i.e. whose contents and metadata are entirely in the kernel page cache) ought to very closely match the curve of those of the idealised random access storage device. This makes this idealisation useful to this specification. In some implementations e.g. those using persistent memory, the storage device *is* your random access memory, and thus by definition the two are quite literally the same thing.

It is therefore important to note in the specification wording below that for reads, it is from the idealised storage device that is specified. If the entity being read is not fully warm cached, the wording (or lack thereof) giving latency guarantees does not apply.

[*Note:* It is unknowable whether a specific read from a file just immediately read from will be warm cached. For example, in a high memory pressure situation, the data just read and cached may be evicted by the time the second read occurs, or a concurrent update may have invalidated the cached region. – end note]

6.5.4 Genericity in i/o

[llfio.principles.io_genericity]

For each of the i/o types, a `buffer_type`, `buffers_type` and `io_request<buffers_type>` is defined as a member type or type alias. Each of the i/o type's `read()` functions will consume an `io_request<buffers_type>`, and return a `buffers_type` (for `write()`, the corollary is `io_request<const_buffers_type>`, and return a `const_buffers_type`).

This design principle is held consistent throughout. Intuitively, a file handle will have a buffer type of `span<byte>`, a buffers type of `span<buffer_type>`, and an `io_request<buffers_type>` type which takes the desired offset to do the i/o at. A directory handle will have a buffer type of `path_view`, a buffers type of `span<buffer_type>` and an `io_request<buffers_type>` type which supplies the directory enumeration options desired e.g. wildcards. A symlink handle will have a buffer type of `path_view`, a buffers type of `path_view` and an `io_request<buffers_type>` type which is nothing more than a `path_view`, as after all a symbolic link can represent exactly one path.

This uniformity of design eases cognitive load on the programmer, and may aid generic code to work with arbitrary i/o capable types, even very different ones. For example, a vector math i/o handle might have a buffer type of `float`, a buffers type of `span<buffer_type>` and an `io_request<buffers_type>` type which says which bulk math operation to do e.g. add. A scalable vector math implementation, such as ARM SVE, would perform as much bulk math as there is currently spare resources on the CPU's or GPU's vector math units, and return those buffers filled with results. A loop can then iterate through each partial buffer fill until the entire operation is completed, exactly the same as with non-blocking file or socket i/o.

6.5.5 Race free filesystem

[`llfio.principles.race_free`]

POSIX.1.2008 standardised a suite of kernel system calls which permit use of the filesystem without risk of race conditions caused by concurrent third party modification of paths. At the time of writing, these *race free filesystem* extensions are very widely implemented in the major operating systems, and given their considerable benefit to writing secure code, we standardise these facilities into C++.

This specification discourages the use of absolute filesystem paths, as these are inherently racy. All manipulation operations such as rename or unlink are performed on an open `handle` instance. If this specification is accepted into the standard, is recommended that all functions for accessing the file system in `[fs.op]` which take an absolute path ought to be deprecated, and their further use discouraged by complete removal in the following standard.

This specification reimplements directory enumeration via `directory_handle::read()` using a mechanism very different to `[fs.class.directory_iterator]`, specifically one without the many correctness and performance problems which have been reported by users on Stack Overflow and other such forums. It is recommended that `filesystem::directory_entry` and `filesystem::directory_iterator` et al be rewritten to use this specification's facilities instead as a backwards compatibility measure, and that new code ought to use this specification instead.

6.6 Header `<io/algorithm/cached_parent_handle_adapter>` `fio.io.algorithm.cached_parent_handle_adapter`

[`ll-`

Todo

6.7 Header `<io/algorithm/shared_fs_mutex>` [ll- io.io.algorithm.shared_fs_mutex]

Todo

6.8 Header `<io/deadline>` [llfio.io.deadline]

6.8.1 Synopsis [llfio.io.deadline.synopsis]

```
1 namespace std { namespace experimental { namespace io { inline namespace v1 {  
2 struct deadline  
3 {  
4     constexpr deadline() noexcept;  
5     constexpr explicit operator bool() const noexcept;  
6     template <class Clock, class Duration>  
7     constexpr deadline(chrono::time_point<Clock, Duration> tp) noexcept;  
8     template <class Rep, class Period>  
9     constexpr deadline(chrono::duration<Rep, Period> d) noexcept;  
10 };  
11 } } } }
```

6.8.2 Class `deadline` [io.deadline.deadline]

A time deadline in either relative-to-now or absolute (system clock) terms. ABI stable, unlike the templated `chrono::duration<>` or `chrono::time_point<>`.

Remarks: The type `deadline` must meet the `TriviallyCopyable` and `StandardLayout` concepts. The internal storage must be of at least nanosecond resolution, but also able to store the full range of dates that `time_t` can store¹⁶.

6.8.2.1 Class `deadline` constructors [llfio.io.deadline.deadline.constructors]

```
constexpr deadline() noexcept;
```

Effects: Constructs an invalid deadline.

Ensures: `!*this`

```
template <class Clock, class Duration>  
constexpr deadline(chrono::time_point<Clock, Duration> tp) noexcept;
```

Effects: Implicitly constructs either an absolute (`Clock::is_steady == false`) or duration-from-now (`Clock::is_steady == true`) deadline.

Ensures: `*this`

¹⁶A union of a struct `timespec` (absolute) and a 64-bit nanosecond count (relative) is suggested, but not required.

```
template <class Rep, class Period>
constexpr deadline(chrono::duration<Rep, Period> d);
```

Effects: Implicitly constructs a duration-from-now deadline.

Ensures: *this

6.9 Header <io/directory_handle> [llfio.io.directory_handle]

Todo

6.10 Header <io/embedded_file_handle> [llfio.io.embedded_file_handle]

Todo

6.11 Header <io/embedded_file_source> [llfio.io.embedded_file_source]

Todo

6.12 Header <io/file_handle> [llfio.io.file_handle]

Todo

6.13 Header <io/handle> [llfio.io.handle]

6.13.1 Synopsis [llfio.io.handle.synopsis]

```
1 namespace std { namespace experimental { namespace io { inline namespace v1 {
2
3 class handle
4 {
5 public:
6     ///! The path type used by this handle
7     using path_type = filesystem::path;
8     ///! The file extent type used by this handle
9     using extent_type = unsigned long long;
10    ///! The memory extent type used by this handle
11    using size_type = size_t;
12
13    ///! The behaviour of the handle: does it read, read and write, or atomic append?
14    enum class mode : unsigned char // bit 0 set means writable
15    {
16        unchanged = 0,
17        none = 2,      ///!< No ability to read or write anything, but can synchronise (SYNCHRONIZE or 0)
18        attr_read = 4, ///!< Ability to read attributes (FILE_READ_ATTRIBUTES|SYNCHRONIZE or 0_RDONLY)
19        attr_write = 5, ///!< Ability to read and write attributes
20        ///!< (FILE_READ_ATTRIBUTES|FILE_WRITE_ATTRIBUTES|SYNCHRONIZE or 0_RDONLY)
```

```

21     read = 6,          //!< Ability to read
22                   //!< (READ_CONTROL|FILE_READ_DATA|FILE_READ_ATTRIBUTES|FILE_READ_EA|SYNCHRONISE
                   or O_RDONLY)
23     write = 7,       //!< Ability to read and write
24                   //!< (READ_CONTROL|FILE_READ_DATA|FILE_READ_ATTRIBUTES|FILE_READ_EA|
                   FILE_WRITE_DATA|FILE_WRITE_ATTRIBUTES|FILE_WRITE_EA|FILE_APPEND_DATA|
                   SYNCHRONISE or O_RDWR)
25     append = 9      //!< All mainstream OSs and CIFS guarantee this is atomic with respect to all
26                   //!< other appenders (FILE_APPEND_DATA|SYNCHRONISE or O_APPEND)
27 };
28
29 //!< On opening, do we also create a new file or truncate an existing one?
30 enum class creation : unsigned char
31 {
32     open_existing = 0, //!< Filesystem entry must already exist
33     only_if_not_exist, //!< Filesystem entry must NOT exist, and is atomically created by the success
                       of this operation
34     if_needed,        //!< If filesystem entry exists that is used, else one is created
35     truncate          //!< Filesystem entry must already exist. It is atomically truncated on open,
                       leaving creation date and unique identifier unmodified.
36 };
37
38 //!< What i/o on the handle will complete immediately due to kernel caching
39 enum class caching : unsigned char // bit 0 set means safety fsyncs enabled
40 {
41     unchanged = 0,
42     none = 1,          //!< No caching whatsoever, all reads and writes come from storage
43                       //!< (i.e. 'O_DIRECT|O_SYNC'). Align all i/o to 4Kb boundaries for this
44                       to work. 'flag_disable_safety_fsyncs' can be used here.
45     only_metadata = 2, //!< Cache reads and writes of metadata but avoid caching data
46                       //!< ('O_DIRECT'), thus i/o here does not affect other cached data for
47                       other handles. Align all i/o to 4Kb boundaries for this to work.
48     reads = 3,        //!< Cache reads only. Writes of data and metadata do not complete
49                       //!< until reaching storage ('O_SYNC'). 'flag_disable_safety_fsyncs' can
50                       be used here.
51     reads_and_metadata = 5, //!< Cache reads and writes of metadata, but writes of data do not
52                               //!< complete until reaching storage ('O_DSYNC'). '
53                               flag_disable_safety_fsyncs' can be used here.
54     all = 4,          //!< Cache reads and writes of data and metadata so they complete
55                       //!< immediately, sending writes to storage at some point when the kernel
56                       decides (this is the default file system caching on a system).
57     safety_fsyncs = 7, //!< Cache reads and writes of data and metadata so they complete
58                       //!< immediately, but issue safety fsyncs at certain points. See
59                       documentation for 'flag_disable_safety_fsyncs'.
60     temporary = 6     //!< Cache reads and writes of data and metadata so they complete
61                       //!< immediately, only sending any updates to storage on last handle
62                       close in the system or if memory becomes tight as this file is
63                       expected to be temporary (Windows and FreeBSD only).
64 };
65
66 //!< Bitwise flags which can be specified
67 bitfield(flag)
68 {
69     none = 0, //!< No flags
70
71     /*! Unlinks the file on handle close. On POSIX, this simply unlinks whatever is pointed

```

```

64 to by 'path()' upon the call of 'close()' if and only if the inode matches. On Windows,
65 if you are on Windows 10 1709 or later, exactly the same thing occurs. If on previous
66 editions of Windows, the file entry does not disappears but becomes unavailable for
67 anyone else to open with an 'errc::resource_unavailable_try_again' error return. Because
68 this is confusing, unless the 'win_disable_unlink_emulation' flag is also specified, this
69 POSIX behaviour is somewhat emulated on older Windows by renaming the file to a random
70 name on 'close()' causing it to appear to have been unlinked immediately.
71 */
72 unlink_on_first_close = 1U << 0U,
73
74 /*! Some kernel caching modes have unhelpfully inconsistent behaviours
75 in getting your data onto storage, so by default unless this flag is
76 specified we add extra fsyncs to the following operations for the
77 caching modes specified below:
78     - truncation of file length either explicitly or during file open.
79     - closing of the handle either explicitly or in the destructor.
80
81 Additionally on Linux only to prevent loss of file metadata:
82     - On the parent directory whenever a file might have been created.
83     - On the parent directory on file close.
84
85 This only occurs for these kernel caching modes:
86     - caching::none
87     - caching::reads
88     - caching::reads_and_metadata
89     - caching::safety_fsyncs
90 */
91 disable_safety_barriers = 1U << 2U,
92
93 /*! 'file_handle::unlink()' could accidentally delete the wrong file if someone has
94 renamed the open file handle since the time it was opened. To prevent this occurring,
95 where the OS doesn't provide race free unlink-by-open-handle, we compare the inode of
96 the path we are about to unlink with that of the open handle before unlinking.
97 Setting this flag disables that safety check.
98 */
99 disable_safety_unlinks = 1U << 3U,
100
101 /*! Ask the OS to disable prefetching of data. This can improve random
102 i/o performance.
103 */
104 disable_prefetching = 1U << 4U,
105
106 /*! Ask the OS to maximise prefetching of data, possibly prefetching the entire file
107 into kernel cache. This can improve sequential i/o performance.
108 */
109 maximum_prefetching = 1U << 5U,
110
111 win_disable_unlink_emulation = 1U << 24U, //!< See the documentation for 'unlink_on_first_close'
112
113 /*! Microsoft Windows NTFS, having been created in the late 1980s, did not originally
114 implement extents-based storage and thus could only represent sparse files via
115 efficient compression of intermediate zeros. With NTFS v3.0 (Microsoft Windows 2000),
116 a proper extents-based on-storage representation was added, thus allowing only 64Kb
117 extent chunks written to be stored irrespective of whatever the maximum file extent
118 was set to.
119

```

```

120     For various historical reasons, extents-based storage is disabled by default in newly
121     created files on NTFS, unlike in almost every other major filing system. You have to
122     explicitly "opt in" to extents-based storage.
123
124     As extents-based storage is nearly cost free on NTFS, llfio by default opts in to
125     extents-based storage for any empty file it creates. If you don't want this, you
126     can specify this flag to prevent that happening.
127     */
128     win_disable_sparse_file_creation = 1U << 25U,
129
130     /*! Filesystems tend to be embarrassingly parallel for operations performed to different
131     inodes. Where LLFIO performs i/o to multiple inodes at a time, it will use OpenMP or
132     the Parallelism or Concurrency standard library extensions to usually complete the
133     operation in constant rather than linear time. If you don't want this default, you can
134     disable default using this flag.
135     */
136     disable_parallelism = 1U << 26U,
137
138     // The following are INFORMATIONAL flags, not set by the user
139     overlapped = 1U << 28U,          /*!< On Windows, create any new handles with OVERLAPPED semantics
140     byte_lock_insanity = 1U << 29U, /*!< Using insane POSIX byte range locks
141     anonymous_inode = 1U << 30U     /*!< This is an inode created with no representation on the
142     filing system
143
144     }
145
146     public:
147     /*! Constructs an invalid instance
148     constexpr handle() noexcept {} // not default to force constexpr checking
149
150     /*! Adopts a supplied native handle type
151     explicit constexpr handle(native_handle_type h, caching caching = caching::none, flag flags = flag::
152     none) noexcept;
153
154     /*! Closes the handles if it is valid
155     virtual ~handle() noexcept;
156
157     // Copy construction is expensive, use clone()
158     handle(const handle &) = delete;
159     handle &operator=(const handle &) = delete;
160
161     // Moves of handle relocate in memory
162     constexpr handle(handle &&o) noexcept = relocates; // from P1029
163     handle &operator=(handle &&o) noexcept;
164     void swap(handle &o) noexcept;
165
166     /*! Retrieve the current path of the open handle
167     virtual path_type current_path() const throws(file_io_error) [[no_side_effects]]; // forthcoming
168     WiP paper
169
170     /*! Close the handle
171     virtual void close() throws(file_io_error);
172     /*! Clone the handle
173     handle clone() const throws(file_io_error);
174     /*! Release the native handle type from management
175     virtual native_handle_type release() noexcept;

```

```

173  //!< True if the handle is valid (and usually open)
174  bool is_valid() const noexcept;
175  //!< True if the handle is readable
176  bool is_readable() const noexcept;
177  //!< True if the handle is writable
178  bool is_writable() const noexcept;
179  //!< True if the handle is append only
180  bool is_append_only() const noexcept;
181  //!< Changes whether this handle is append only or not.
182  virtual void set_append_only(bool enable) throws(file_io_error);
183
184  //!< True if overlapped
185  bool is_overlapped() const noexcept;
186  //!< True if seekable
187  bool is_seekable() const noexcept;
188  //!< True if requires aligned i/o
189  bool requires_aligned_io() const noexcept;
190
191  //!< True if a regular file or device
192  bool is_regular() const noexcept;
193  //!< True if a directory
194  bool is_directory() const noexcept;
195  //!< True if a symlink
196  bool is_symlink() const noexcept;
197  //!< True if a memory section
198  bool is_section() const noexcept;
199
200  //!< Kernel cache strategy used by this handle
201  caching kernel_caching() const noexcept;
202  //!< True if the handle uses the kernel page cache for reads
203  bool are_reads_from_cache() const noexcept;
204  //!< True if writes are safely on storage on completion
205  bool are_writes_durable() const noexcept;
206  //!< True if issuing safety fsyncs is on
207  bool are_safety_barriers_issued() const noexcept;
208
209  //!< The flags this handle was opened with
210  flag flags() const noexcept;
211  //!< The native handle used by this handle
212  native_handle_type native_handle() const noexcept;
213 };
214 inline std::ostream &operator<<(std::ostream &s, const handle &v);
215 inline std::ostream &operator<<(std::ostream &s, const handle::mode &v);
216 inline std::ostream &operator<<(std::ostream &s, const handle::creation &v);
217 inline std::ostream &operator<<(std::ostream &s, const handle::caching &v);
218 inline std::ostream &operator<<(std::ostream &s, const handle::flag &v);
219 } } } }

```

6.13.2 Class `handle`

[`llfio.io.handle.handle`]

An `io::native_handle_type` instance, whose lifetime is managed by the lifetime of this object.

Remarks: The type `handle` must meet the `MoveRelocating` concept.

6.13.2.1 Class `handle` constructors and destructor [`llfio.io.handle.handle.constructors`]

```
constexpr handle() noexcept {}
```

Effects: Constructs an invalid `handle`. Must be in-class defined for [`P1029`] *move = relocates*.

Ensures: `!is_valid()`

```
explicit constexpr handle(native_handle_type h, caching caching = caching::none,  
                          flag flags = flag::none) noexcept;
```

Effects: Adopt a `native_handle_type` instance.

Ensures: `is_valid()`

```
constexpr handle(handle &&o) noexcept = relocates;
```

Effects: `Relocate` moves the management of the `native_handle_type` out of `o` and into `*this`. Implemented using [`P1029`] *move = relocates*.

Ensures: `!o.is_valid()`

```
virtual ~handle() noexcept;
```

Effects: If this `handle` is valid, call `close()`. If `close()` fails, terminate the process. Guaranteed that if called on a default constructed instance, it will have no side effects (which includes any derived classes, as per `P1029` *relocates* requirements).

Ensures: `!is_valid()`, and that the system resources managed by the `handle` are released.

6.13.2.2 Class `handle` assignment and swap [`llfio.io.handle.handle.assignments`]

```
handle &operator=(handle &&o) noexcept;
```

Effects: If this `handle` is valid, call `close()`. If `close()` fails, terminate the process. Then move the management of the `native_handle_type` out of `o` and into `*this`.

Ensures: `!o.is_valid()`, and that the system resources formerly managed by the `handle` are released.

```
void swap(handle &o) noexcept;
```

Effects: Exchange the management of the two `native_handle_type` between `o` and `*this`.

6.13.2.3 Class `handle` observers [`llfio.io.handle.handle.observers`]

```
virtual path_type current_path() const throws(file_io_error) [[no_side_effects]];
```

Effects: Returns the current path of the hard link originally opened by this handle, as is said by the operating system. If the hard link originally opened by this handle has been unlinked, an empty path is returned. If somebody with an open handle to the same hard link which has been unlinked then relinks it, the new path is returned. Calls to this function have no side effects, and may be elided by the compiler.

Remarks: The path returned may be very different to the path by which the handle was originally opened (e.g. due to a third party renaming it, or due to canonicalisation of a symbolically linked input path), but it must be to the original hard link opened, and not to some arbitrary path which links to the same inode. This implies that either the kernel, or the standard library implementation, must implement *hard link tracking* by keeping with each kernel file descriptor which hard link it was opened with. At the time of writing, the Linux and Microsoft Windows kernels fully implement this; FreeBSD has a partial implementation; Apple MacOS does not implement this. It is permitted on kernels without complete kernel support to implement hard link tracking in the standard library using a shared memory region – in this case, only programs using standard C++ library facilities to work with the filing system would have defined behaviour.

[*Note:* It is suggested that if your operating system does not fully implement hard link tracking, its kernel ought to be fixed in preference to emulating the support using shared memory. – end note]

Throws: Many possible causes of failure, including failure to allocate memory, any failure returned by the kernel (that the handle does not refer to a resource with a path, that the handle is invalid), etc.

Complexity: This is a latency degrading function, which may make several kernel syscalls, allocate a number of items from memory, perform unbounded loops of checks of properties of filesystem entities, and perform other non-deterministic processing, including waits to hold multiple kernel mutexes.

```
bool is_valid() const noexcept;
```

Returns: True if this handle is managing a valid native handle type.

```
bool is_readable() const noexcept;
```

Returns: True if this handle is managing a native handle type which can be read from.

```
bool is_writable() const noexcept;
```

Returns: True if this handle is managing a native handle type which can be written to.

```
bool is_append_only() const noexcept;
```

Returns: True if this handle is managing a native handle type which can be written to, but only to append to whatever its current maximum extent is.

`bool is_seekable() const noexcept;`

Returns: True if this handle observes the offset specified during a read or write, false if the offset is ignored.

`bool requires_aligned_io() const noexcept;`

Returns: True if this handle requires reads and writes to be performed on a storage device determined alignment, and in round multiples of that alignment.

`bool is_regular() const noexcept;`

Returns: True if this handle represents a regular file or device.

`bool is_directory() const noexcept;`

Returns: True if this handle represents a directory.

`bool is_symlink() const noexcept;`

Returns: True if this handle represents a symbolic link.

`bool is_section() const noexcept;`

Returns: True if this handle represents a mappable section of memory.

`caching kernel_caching() const noexcept;`

Returns: The caching value this handle was constructed with.

`bool are_reads_from_cache() const noexcept;`

Returns: True if reads from this handle come from kernel page cache memory.

`bool are_writes_durable() const noexcept;`

Returns: True if writes to this handle do not complete until all of the data written is wholly upon the storage device.

`bool are_safety_barriers_issued() const noexcept;`

Returns: True if safety flushes of metadata to storage are automatically issued by the library.

`flag flags() const noexcept;`

Returns: The flags value this handle was constructed with.

```
native_handle_type native_handle() const noexcept;
```

Returns: The native handle type managed by this handle instance.

6.13.2.4 Class `handle` modifiers

[llfio.io.handle.handle.modifiers]

```
virtual void close() throws(file_io_error);
```

Effects: If this handle is valid, close the managed native handle type with the kernel, releasing any resources.

Ensures: `!is_valid()`, and that the system resources formerly managed by the handle are released.

Throws: Any failure returned by the kernel.

```
handle clone() const throws(file_io_error);
```

Effects: If this handle is valid, duplicates the managed native handle type with the kernel. If this handle is not valid, returns a similarly invalid handle instance.

Throws: Any failure returned by the kernel.

```
virtual native_handle_type release() noexcept;
```

Effects: Release the managed native handle type from management by this handle instance.

Ensures: `!is_valid()`.

```
virtual void set_append_only(bool enable) throws(file_io_error);
```

Effects: Sets whether writes to this handle (atomically) append to whatever its current maximum extent is, or whether the offset specified during the write is used.

Ensures: `is_append_only() == enable, is_seekable() == !enable`

6.14 Header `<io/io_handle>`

[llfio.io.io_handle]

6.14.1 Synopsis

[llfio.io.io_handle.synopsis]

```
1 namespace std { namespace experimental { namespace io { inline namespace v1 {
2
3   class io_handle : public handle
4   {
5   public:
6     using path_type = handle::path_type;
```

```

7   using extent_type = handle::extent_type;
8   using size_type = handle::size_type;
9   using mode = handle::mode;
10  using creation = handle::creation;
11  using caching = handle::caching;
12  using flag = handle::flag;
13
14  ///! The scatter buffer type used by this handle.
15  ///! Guaranteed to be 'TrivialType' and 'StandardLayoutType' and to match
16  ///! in layout 'struct iovec' on POSIX. Quacks like span<byte>.
17  struct buffer_type;
18  ///! The gather buffer type used by this handle.
19  ///! Guaranteed to be 'TrivialType' and 'StandardLayoutType' and to match
20  ///! in layout 'struct iovec' on POSIX. Quacks like span<const byte>.
21  struct const_buffer_type;
22
23  ///! The scatter buffers type used by this handle.
24  ///! Guaranteed to be 'TrivialType' apart from construction, and 'StandardLayoutType'.
25  using buffers_type = span<buffer_type>;
26  ///! The gather buffers type used by this handle.
27  ///! Guaranteed to be 'TrivialType' apart from construction, and 'StandardLayoutType'.
28  using const_buffers_type = span<const_buffer_type>;
29
30  ///! The i/o request type used by this handle.
31  ///! Guaranteed to be 'TrivialType' apart from construction, and 'StandardLayoutType'.
32  template <class T> struct io_request
33  {
34      T buffers{};
35      extent_type offset{0};
36
37      ///! Default constructor
38      io_request() = default;
39      ///! Implicit construction from a set of buffers, and an offset
40      constexpr io_request(T _buffers, extent_type _offset);
41  };
42
43 public:
44     ///! Default constructor
45     constexpr io_handle() {} // Not default to force constexpr check
46     ~io_handle() override = default;
47
48     ///! Construct a handle from a supplied native handle
49     constexpr explicit io_handle(native_handle_type h, caching caching = caching::none, flag flags =
50         flag::none) noexcept;
51     ///! Explicit conversion from handle permitted
52     explicit constexpr io_handle(handle &&o) noexcept;
53     ///! Move construction permitted
54     io_handle(io_handle &&) = relocates;
55     ///! No copy construction (use 'clone()')
56     io_handle(const io_handle &) = delete;
57     ///! Move assignment permitted
58     io_handle &operator=(io_handle &&) = default;
59     ///! No copy assignment
60     io_handle &operator=(const io_handle &) = delete;
61
62     /*! \brief The *maximum* number of buffers which a single read or write syscall can process

```

```

62 at a time for this specific open handle.
63 */
64 virtual size_t max_buffers() const noexcept [[no_side_effects]];
65
66 ///! Read data from the open handle.
67 virtual buffers_type read(io_request<buffers_type> reqs, deadline d = deadline()) throws(
68     file_io_error)
69     [[no_side_effects]];
70
71 ///! Write data to the open handle.
72 virtual const_buffers_type write(io_request<const_buffers_type> reqs, deadline d = deadline())
73     throws(file_io_error);
74
75 ///! Convenience overload taking an initialiser list
76 size_type write(extent_type offset, initializer_list<const_buffer_type> lst, deadline d = deadline()
77     ) throws(file_io_error);
78
79 ///! \brief Issue a write reordering barrier such that writes preceding the barrier will reach
80 storage before writes after this barrier.
81 */
82 virtual const_buffers_type barrier(io_request<const_buffers_type> reqs = {},
83     bool wait_for_device = false, bool and_metadata = false,
84     deadline d = deadline()) throws(file_io_error) = 0
85     [[no_visible_side_effects]];
86
87 ///! \class extent_guard
88 \brief RAII holder a locked extent of bytes in a file.
89 */
90 class extent_guard
91 {
92 public:
93     extent_guard(const extent_guard &) = delete;
94     extent_guard &operator=(const extent_guard &) = delete;
95
96     ///! Default constructor
97     constexpr extent_guard() {}
98     ///! Move constructor
99     extent_guard(extent_guard &&o) noexcept = relocates;
100     ///! Move assign
101     extent_guard &operator=(extent_guard &&o) noexcept;
102     ~extent_guard();
103     ///! True if extent guard is valid
104     explicit operator bool() const noexcept;
105     ///! True if extent guard is invalid
106     bool operator!() const noexcept;
107
108     ///! The io_handle to be unlocked
109     io_handle *handle() const noexcept;
110     ///! Sets the io_handle to be unlocked
111     void set_handle(io_handle *h) noexcept;
112     ///! The extent to be unlocked
113     std::tuple<extent_type, extent_type, bool> extent() const noexcept;
114
115     ///! Unlocks the locked extent immediately
116     void unlock() noexcept;

```

```

115     ///! Detach this RAII unlocker from the locked state
116     void release() noexcept;
117 };
118
119 /*! \brief Tries to lock the range of bytes specified for shared or exclusive access. Be aware this
120 passes through the same semantics as the underlying OS call, including any POSIX insanity present on
121 your platform:
122 - Any fd closed on an inode must release all byte range locks on that inode for all
123 other fds. If your OS isn't new enough to support the non-insane lock API,
124 'flag::byte_lock_insanity' will be set in flags() after the first call to this function.
125 - Threads replace each other's locks, indeed locks replace each other's locks.
126 You almost certainly should use your choice of an 'algorithm::shared_fs_mutex::*' instead of this
127 as those are more portable and performant.
128 \warning This is a low-level API which you should not use directly in portable code. Another issue
129 is that atomic lock upgrade/downgrade, if your platform implements that (you should assume
130 it does not in portable code), means that on POSIX you need to *release* the old 'extent_guard'
131 after creating a new one over the same byte range, otherwise the old 'extent_guard''s
132 destructor will simply unlock the range entirely. On Windows however upgrade/downgrade
133 locks overlay, so on that platform you must *not* release the old
134 'extent_guard'. Look into 'algorithm::shared_fs_mutex::safe_byte_ranges' for a portable solution.
135 \return An extent guard, the destruction of which will call unlock().
136 \param offset The offset to lock. Note that on POSIX the top bit is always cleared before use
137 as POSIX uses signed transport for offsets. If you want an advisory rather than mandatory lock
138 on Windows, one technique is to force top bit set so the region you lock is not the one you will
139 i/o - obviously this reduces maximum file size to (2^63)-1.
140 \param bytes The number of bytes to lock. Zero means lock the entire file using any more
141 efficient alternative algorithm where available on your platform (specifically, on BSD and OS X use
142 flock() for non-insane semantics).
143 \param exclusive Whether the lock is to be exclusive.
144 \param d An optional deadline by which the lock must complete, else it is cancelled.
145 \errors Any of the values POSIX fcntl() can return, 'errc::timed_out', 'errc::not_supported' may be
146 returned if deadline i/o is not possible with this particular handle configuration (e.g.
147 non-overlapped HANDLE on Windows).
148 \mallocs The default synchronous implementation in file_handle performs no memory allocation.
149 The asynchronous implementation in async_file_handle performs one calloc and one free.
150 */
151 virtual extent_guard lock(extent_type offset, extent_type bytes, bool exclusive = true, deadline d =
152     deadline()) throws(file_io_error) [[no_visible_side_effects]];
153
154 extent_guard try_lock(extent_type offset, extent_type bytes, bool exclusive = true) throws(
155     file_io_error) [[no_visible_side_effects]];
156
157 /*! \overload Locks for shared access
158 extent_guard lock(io_request<buffers_type> reqs, deadline d = deadline()) throws(file_io_error) [[
159     no_visible_side_effects]];
160
161 /*! \overload Locks for exclusive access
162 extent_guard lock(io_request<const_buffers_type> reqs, deadline d = deadline()) throws(file_io_error
163     ) [[no_visible_side_effects]];
164
165 /*! \brief Unlocks a byte range previously locked.
166 \param offset The offset to unlock. This should be an offset previously locked.
167 \param bytes The number of bytes to unlock. This should be a byte extent previously locked.
168 \errors Any of the values POSIX fcntl() can return.
169 \mallocs None.
170 */
171 virtual void unlock(extent_type offset, extent_type bytes) noexcept [[no_visible_side_effects]];

```

```
167 };
168
169 } } } }
```

6.14.2 Class `io_handle` [llfio.io.io_handle.io_handle]

A refinement of `handle` capable of scatter-gather reading and writing data, as well as advisory locking regions of data for shared or exclusive access. As the base handle type for all specific implementations of i/o handle, using this abstracted base class for i/o is considerably more complex to get correct than using a specific implementation type, where i/o behaviour guarantees are more firmly set. Normative guidance notes for correct use is therefore given in some cases.

Remarks: The type `io_handle` must meet the `MoveRelocating` concept.

6.14.2.1 Class `io_handle` constructors [llfio.io.io_handle.io_handle.constructors]

```
constexpr explicit io_handle(native_handle_type h, caching caching = caching::none,
                             flag flags = flag::none) noexcept;
```

Effects: Adopt a `native_handle_type` instance.

Ensures: `is_valid()`

6.14.2.2 Class `io_handle` observers [llfio.io.handle.handle.observers]

```
virtual size_t max_buffers() const noexcept [[no_side_effects]];
```

Returns: The *maximum* number of i/o buffers which a single read or write function can *atomically* process at a time for this specific open handle. This enables programs to not prepare longer scatter-gather buffers than the system is capable of.

[*Note:* The *actual* maximum number of i/o buffers which a single read or write function can process at a time is dependent upon available system resources at the time of the call. As an example of how low they can be, MAC OS permits a maximum of sixteen asynchronous i/o operations to be in flight per process. – end note]

[*Note:* If this function returns 1, scatter-gather i/o is implemented as a loop over the supplied buffer list. In this situation, there is no particular benefit to supply more than one buffer per operation, apart from programmer convenience. – end note]

6.14.2.3 Class `io_handle` modifiers [llfio.io.handle.handle.modifiers]

```
virtual buffers_type read(io_request<buffers_type> reqs,
                          deadline d = deadline()) throws(file_io_error)
    [[no_side_effects]];
```

Effects: For each buffer, if the address of the buffer returned is that of the buffer supplied:

- Input buffers are filled to no further than their size on input, with data read from the open handle at the offset supplied by the caller (if the open handle supports offset seeks), proceeding byte by byte from that offset onwards.
- If the handle was opened with `mode::append` i.e. `is_append_only()` is true, behaviour is implementation defined.
- As the individual buffers returned are the individual buffers input which were filled (i.e. the array of `buffer_type` pointed to by `buffers_type`), you must ensure that the individual buffers input you supply are safe to write to.
- The number of buffers filled are those returned, which may be less than the number of buffers input. Input buffers not referenced by buffers returned will not be modified.
- At least the first byte of the first of the non-zero sized input buffers will be filled if the function returns successfully.
- Buffers may be filled in any order, but for the buffers returned, every buffer is completely filled up until the final partially filled buffer (if any).
- If the last buffer of those returned is not completely filled, its size in the returned buffers will be set to the partial bytes filled into that buffer.
- If the last buffer of those to be returned is fully filled, and not all input buffers were filled, a zero sized buffer will be placed in the final buffers returned.
- Input buffers can be left inconsistent if an error is thrown.
- If the deadline is default constructed, the function may block, possibly forever, until at least one byte is read, or possibly until all requested bytes are read, or anywhere in between depending on implementation.

If the deadline specifies an absolute deadline, and the system clock passes that deadline, any pending i/o will be cancelled and a partial buffer fill shall be returned if any of at least the first buffer was filled.

If the deadline specifies an elapsed period and the i/o has not completed after that period, any pending i/o will be cancelled a partial buffer fill shall be returned if any of at least the first buffer was filled.

[*Note:* Cancelling pending i/o may be a blocking operation of indeterminate length for some implementations i.e. there is no guarantee that the function will return at a time anywhere close to what you requested. – end note]

If the deadline specifies a zeroed deadline, the implementation will fill as many buffers as it can without blocking, and return immediately.

- Implementations may not implement deadline i/o. If they do not, they will return an error comparing equal to `errc::operation_not_supported` for any deadline which is not default constructed.
- Implementations may only implement non-timed deadline i/o i.e. deadline must be default constructed, or zeroed, and nothing else. In this situation, an error comparing equal to

`errc::operation_not_supported` will be returned.

[*Note:* This situation arises where an implementation must call extra syscalls to implement timed deadline i/o which breaks the latency preserving guarantee of this `read()` function. As implementations of this function cannot add complexity over the underlying syscall, if there is only a choice of blocking or non-blocking syscall, implementations must use those and not add extra syscalls (e.g. `select()`) to implement timed i/o expiry. – end note]

If the address of the buffer returned is not that of the buffer supplied:

- The address and size returned points to data which would have been filled had the buffers input been filled, same as above.
- You may not write into the buffer in this situation as the memory does not belong to you.
- You are guaranteed that the memory region returned to you will exist as long as this handle instance remains valid.

[*Note:* It is possible for implementations to return a mixture of filled buffers and buffers pointing elsewhere. If you need to test that your code using `io_handle` works correctly, `fast_random_file_handle` can be used to return random mixes of filled buffers input and buffers pointing elsewhere. – end note]

Other things to consider:

- Not all buffers supplied may be filled during the call, even if the data is available, and may require unlimited subsequent calls to complete the original scatter fill request. In this situation, you should adjust the buffers input based on the buffers returned, and perform the operation again, looping this procedure until all original input buffers are filled. See below for an illustrative example.
- It is implementation defined whether concurrent reads of one or more buffers may see partial completion of any concurrent writes to the same offset and extent. In particular, a concurrent write may not update in a linear lower to upper offset fashion – it may appear to update a later offset first, then an earlier one, then a middle one, in any arbitrary (i.e. unpredictable) order.
- It is implementation defined if a read exceeds any current maximum extent of the storage referenced by the handle. It may clamp buffers returned to that maximum extent (i.e. by partially filling the last fillable buffer, and marking all remaining buffers as being unfilled), or fill with garbage instead, or clamp to a different maximum extent, or any other behaviour.
- It is implementation defined whether any system current file pointer for the open handle is affected by this operation.
- If you perform a read after a write, only fully completed writes to the same handle instance you read from are guaranteed to be wholly observable to the read. It is implementation defined whether, or when, writes to other handles to the same resource will become visible to reads from a separate handle instance.

- It is implementation defined if a single `io_handle` instance may be used by multiple threads, even if those threads mutually exclude concurrent usage (some implementations may be hard-wired into their creating thread, and misoperate if ever called by a different thread).

Throws: There are multiple causes of failure: (i) any failure returned by the kernel (that the handle is invalid, that the handle is not open for reading, that insufficient system resources are available to perform the read, that the operation was cancelled by another thread etc) (ii) if a non-infinite deadline is supplied and this handle does not support deadline i/o, may fail with an error comparing equal to `errc::function_not_supported` (iii) if a non-infinite deadline is supplied and that deadline expired, an error comparing equal to `errc::timed_out` will be thrown (it is implementation defined if the buffers input were read into, or if the buffers input have been updated to indicate what buffers were filled).

Complexity: Depending on implementation, this may be a latency preserving function in the warm cache use case, where the curve of the sorted latency distribution will closely match that of the curve of copying memory at the same offsets over the same extent of memory. However it may also be a latency degrading function which allocates memory, or performs network access of indeterminate duration. At this abstract level in the type hierarchy, you cannot know which. You should use a refinement of this type directly if you want stronger guarantees.

[*Example:* Given the all of the conditions of use just described, it will be non-trivial to use `io_handle::read()` correctly. It is recommended that the first course of action is to use a derived type with much stronger guarantees – such as `file_handle::read()` – where most of the complexity below can be dispensed with. If however you can have no idea what the implementation type actually is, one ends up with code like the below function which will retry individual read operations until a scatter buffer list is filled. Note that the below code may hang forever depending on the `io_handle` implementation, which is why it is not supplied as part of the proposed standard library. It also does not handle large gather buffer lists, which would overflow the stack.

```

1 inline io_handle::buffers_type read_all(io_handle &h, io_handle::io_request<io_handle::
   buffers_type> reqs, deadline d = deadline()) throws(file_io_error)
2 {
3     // Record beginning if deadline is specified
4     chrono::steady_clock::time_point began_steady;
5     if(d && d.steady)
6         began_steady = chrono::steady_clock::now();
7
8     // Take copy of input buffers onto stack, and set output buffers to buffers supplied
9     auto *input_buffers_mem = reinterpret_cast<io_handle::buffer_type *>(alloca(reqs.buffers
   .size() * sizeof(io_handle::buffer_type)));
10    auto *input_buffers_sizes = reinterpret_cast<io_handle::extent_type *>(alloca(reqs.
   buffers.size() * sizeof(io_handle::extent_type)));
11    io_handle::buffers_type output_buffers(reqs.buffers);
12    io_handle::io_request<io_handle::buffers_type> creq({input_buffers_mem, reqs.buffers.
   size()}, 0);
13    for(size_t n = 0; n < reqs.buffers.size(); n++)
14    {
15        // Copy input buffer to stack and retain original size
16        creq.buffers[n] = reqs.buffers[n];
17        input_buffers_sizes[n] = reqs.buffers[n].size();
18        // Set output buffer length to zero

```

```

19     output_buffers[n] = io_handle::buffer_type{output_buffers[n].data(), 0};
20 }
21
22 // Track which output buffer we are currently filling
23 size_t idx = 0;
24 do
25 {
26     // New deadline for this loop
27     deadline nd;
28     if(d)
29     {
30         if(d.steady)
31         {
32             auto ns = chrono::duration_cast<chrono::nanoseconds>((began_steady + chrono::
nanoseconds(d.nsecs)) - chrono::steady_clock::now());
33             if(ns.count() < 0)
34                 nd.nsecs = 0;
35             else
36                 nd.nsecs = ns.count();
37         }
38         else
39             nd = d;
40     }
41     // Partial fill buffers with current request
42     io_handle::buffers_type filled = h.read(creq, nd);
43
44     // Adjust output buffers by what was filled, and prepare input
45     // buffers for next round of partial fill
46     for(size_t n = 0; n < filled.size(); n++)
47     {
48         // Add the amount of this buffer filled to next offset read and to output buffer
49         auto &input_buffer = filled[n];
50         auto &output_buffer = output_buffers[idx + n];
51         creq.offset += input_buffer.size();
52         output_buffer = io_handle::buffer_type{output_buffer.data(), output_buffer.size() +
input_buffer.size()};
53         // Adjust input buffer to amount remaining
54         input_buffer = io_handle::buffer_type{input_buffer.data() + input_buffer.size(),
input_buffers_sizes[idx + n] - output_buffer.size()};
55     }
56
57     // Remove completely filled input buffers
58     while(!creq.buffers.empty() && creq.buffers[0].size() == 0)
59     {
60         creq.buffers = io_handle::buffers_type(creq.buffers.data() + 1, creq.buffers.size()
- 1);
61         ++idx;
62     }
63 } while(!creq.buffers.empty());
64 return output_buffers;
65 }

```

– end example]

```
virtual const_buffers_type write(io_request<const_buffers_type> reqs,
```

```
deadline d = deadline()) throws(file_io_error);
```

Effects:

- For each buffer supplied, write the specified data and amount of data to the specified offset into the open handle (if the open handle supports offset seeks), incrementing the offset by each buffer written in turn.
- If the handle was opened with `mode::append` i.e. `is_append_only()` is true, the specified offset shall be ignored, and instead the maximum extent of the file will be atomically incremented by the sum of the lengths of the buffers up as far as `max_buffers()` at a time, and then the data shall be written into the newly appended extent.
- As the individual buffers returned are the individual buffers input which were filled (i.e. the array of `const_buffer_type` pointed to by `const_buffers_type`), you must ensure that the individual buffers input you supply are safe to write to.
- The number of buffers written are those returned, which may be less than the number of buffers input. Input buffers not referenced by buffers returned will not be modified.
- At least the first byte of the first of the non-zero sized input buffers will be written if the function returns successfully.
- Buffers may be written in any order, but for the buffers returned, every buffer is completely written up until the final partially written buffer (if any).
- If the last buffer of those returned is not completely written, its size in the returned buffers will be set to the partial bytes written from that buffer.
- If the last buffer of those to be returned is fully written, and not all input buffers were written, a zero sized buffer will be placed in the final buffers returned.
- Input buffers can be left inconsistent if an error is thrown.
- If the deadline is default constructed, the function may block, possibly forever, until at least one byte is written, or possibly until all requested bytes are written, or anywhere in between depending on implementation.

If the deadline specifies an absolute deadline, and the system clock passes that deadline, any pending i/o will be cancelled and a partial buffer write shall be returned if any of at least the first buffer was written.

If the deadline specifies an elapsed period and the i/o has not completed after that period, any pending i/o will be cancelled a partial buffer write shall be returned if any of at least the first buffer was written.

If the deadline specifies a zeroed deadline, the implementation will write as many buffers as it can without blocking, and return immediately.

[*Note:* Cancelling pending i/o may be a blocking operation of indeterminate length for some implementations i.e. there is no guarantee that the function will return at a time anywhere close to what you requested. – end note]

Other things to consider:

- Not all buffers supplied may be written during the call, and may require unlimited subsequent calls to complete the original gather write request. In this situation, you should adjust the buffers input based on the buffers returned, and perform the operation again, looping this procedure until all original input buffers are written. See the example code for `read()` for an illustrative example.
- It is implementation defined whether concurrent reads of one or more buffers may see partial completion of any concurrent writes to the same offset and extent. In particular, a concurrent write may not update in a linear lower to upper offset fashion – it may appear to update a later offset first, then an earlier one, then a middle one, in any arbitrary (i.e. unpredictable) order.
- It is implementation defined if a write exceeds any current maximum extent of the storage referenced by the handle. It may clamp buffers returned to that maximum extent (i.e. by partially writing the overlapping buffer, and marking all remaining buffers as being unwritten), or appear to succeed but in fact the data written past the maximum extent is lost, or clamp to a different maximum extent, or any other behaviour. If you wish to guarantee that writes past the maximum extent are safe, open the handle with `mode::append`, or use `set_append_only()` on the handle before the write.
- It is implementation defined whether the order of writes issued by the C++ program to a cached writes configured handle are retained in an order on any underlying storage device i.e. if sudden power loss occurs after a sequence of writes to a handle with cached writes, it is permitted for an implementation to have completely reordered, or partially torn, any of those writes. For handles where write caching was disabled, you are guaranteed that the order of writes is preserved to the storage device, but the metadata to retrieve them may not be. For handles where write and metadata caching was disabled, you are guaranteed that writes form a sequentially consistent ordering fully retrievable in the order they were issued before any unexpected interruption.
- It is implementation defined whether any system current file pointer for the open handle is affected by this operation.

Throws: There are multiple causes of failure: (i) any failure returned by the kernel (that the handle is invalid, that the handle is not open for writing, that insufficient system resources are available to perform the write, that the operation was cancelled by another thread etc) (ii) if a non-infinite deadline is supplied and this handle does not support deadline i/o, may fail with an error comparing equal to `errc::function_not_supported` (iii) if a non-infinite deadline is supplied and that deadline expired, an error comparing equal to `errc::timed_out` will be thrown.

Complexity: Depending on implementation, this may be a latency preserving function in the warm cache use case, where the curve of the sorted latency distribution will closely match that of the curve of copying memory at the same offsets over the same extent of memory. However it may also be a latency degrading function which allocates memory, or performs network access of indeterminate duration. At this abstract level in the type hierarchy, you cannot know which. You should use a refinement of this type directly if you want stronger guarantees.

```
size_type write(extent_type offset, initializer_list<const_buffer_type> lst,
                deadline d = deadline()) throws(file_io_error);
```

Effects: Convenience overload for the preceding `write()` function, this instantiates a stack allocated `io_request<const_buffers_type>` out of the initialiser list supplied, calls the preceding `write()` function, sums the total bytes filled into the buffers returned, and returns that value.

Throws: All the ways in which the preceding `write()` function can fail; also `errc::no_buffer_space` if the size of the input initialiser list exceeds an implementation defined limit (prevents stack overflow).

```
virtual const_buffers_type barrier(io_request<const_buffers_type> reqs = {},
                                  bool wait_for_device = false,
                                  bool and_metadata = false,
                                  deadline d = deadline()) throws(file_io_error)
    [[no_visible_side_effects]] = 0;
```

Effects: For each buffer supplied, do not reorder writes of bytes within those regions across this write barrier. For an empty buffer list, the whole file is assumed.

If `wait_for_device` is true, block the calling thread until all writes to the regions specified have reached the device before returning. This may take some time, but may be relatively fast for some implementations as one need not necessarily wait for acknowledgement that the writes have been persisted, just that they have been received by the device.

If `and_metadata` is true, also block the calling thread until the metadata necessary for retrieving, after sudden power loss, the writes preceding the write barrier to the regions specified have also reached the device before returning. This almost invariably causes a significant blocking wait as one must await acknowledgement that writes, plus metadata, plus any allocations to store them, have been completely persisted.

A non-default deadline permits the call to return with a failure comparing equal to `errc::timed_out` if the writes to the regions specified have not reached the device by the time of the deadline.

[*Note:* In POSIX terms, `wait_for_device` corresponds in terms of semantics to `fdatasync()`, and when combined with `and_metadata`, to `fsync()`, though note that implementations may not actually use these POSIX functions where proprietary functions offer more control. – end note]

Implementations are *completely free* to implement this function as a null operation i.e. to do nothing, yet to report success.

[*Note:* This is required by POSIX, but is also common practice in virtual machine hosts to prevent a single virtual machine causing a denial of service attack to the other virtual machines. End users should very strongly consider opening the handle with `caching::reads` instead, this causes all writes to such a handle to have a sequentially consistent order, and unlike write barriers, if you obtain an open handle with that caching, you are guaranteed that it is working. Note that filing systems use alternative algorithms for files opened with `caching::reads` which often yield *far* superior

performance than using this function. – end note]

Implementations may also choose to always flush metadata when blocking on preceding writes i.e. `and_metadata` is assumed to be always true.

Implementations may ignore the regions supplied for barrier, and may always barrier the whole file instead. In this situation, the buffers returned would be empty to indicate the actual operation carried out.

Implementations may implement non-blocking barriers as blocking barriers (i.e. `wait_for_device` is always true) if the platform does not support non-blocking write barriers¹⁷.

[*Alternative to the above yet to be discussed:* Where an implementation knows for a fact that it cannot implement a requested function because of lacking support on the host platform e.g. to barrier writes without blocking until they reach the device, a failure comparing equal to `errc::function_not_supported` should be returned. – end alternative]

Implementations may be racy with respect to concurrent barriers on overlapping regions by different threads or processes. In this situation, there may be no sequentially consistent ordering, rather an interleaving of regions updated.

Throws: (i) Any failure returned by the kernel (ii) if a non-infinite deadline is supplied and this handle does not support deadline i/o, may fail with an error comparing equal to `errc::function_not_supported` (iii) if a non-infinite deadline is supplied and that deadline expired, an error comparing equal to `errc::timed_out` will be thrown.

Complexity: Depending on implementation, this may be a latency preserving function with a fixed cost of execution. It may also be a latency degrading function of indeterminate execution time.

```
virtual extent_guard lock(extent_type offset,
                        extent_type bytes,
                        bool exclusive = true,
                        deadline d = deadline()) throws(file_io_error)
    [[no_visible_side_effects]];
```

Effects: Tries to place an advisory lock on the range of bytes between `offset` and `bytes` for shared or exclusive access, returning an object of type `extent_guard` whose destruction releases the lock via calling `unlock()` upon the same region. If `bytes` is zero, the whole file is locked¹⁸ i.e. zero up to the maximum possible extent inclusive. Locked regions and allocated extents have nothing to do with one another, so one can lock regions not within any maximum extent.

The effects of this call are permitted to vary widely between implementations. Some known variations:

¹⁷Note to implementers: On impoverished POSIX implementations without better proprietary system calls, a non-blocking barrier can be implemented by launching a `fsync()` task in a separate worker thread, and blocking any further writes on that handle until the worker thread task completes.

¹⁸Some implementations use a completely different mechanism in this situation, though you are guaranteed that whole file and byte region locks see one another.

- All advisory locks may be silently released if **any** handle to the same inode is closed in the process. If the implementation detects this to be the case for a given handle instance, it will set `flags::byte_lock_insanity` so code can detect such unhelpful semantics.

[*Note:* This is the infamous ‘byte lock insanity’ mandated by POSIX. The sooner POSIX fixes such daft semantics, the better. – end note]

- Advisory locks on an inode may silently replace existing locks which overlap any of the same region, including those issued by other threads in the same process to other handles to the same inode.

[*Note:* This unfortunate semantic is also required by POSIX. – end note]

- Some of the top bits of `offset` may be masked off in some implementations, thus making it impossible to take locks on later extents in a file. Implementations are required to return an error comparing equal to `errc::argument_out_of_domain` if this occurs.
- Some of the top bits of `bytes` may be ignored in some implementations, thus limiting the size of the region locked. Implementations are required to return an error comparing equal to `errc::argument_out_of_domain` if this occurs.

[*Note:* The above two are caused by POSIX requiring offset and length to be signed values, and possibility of 31 bit quantities even when files have 64 bit maximum extents. Given that the top bit is always ignored on POSIX, it is suggested that non-POSIX implementations may choose to emulate advisory locks on systems with only mandatory locks by silently forcing the top bit to set, and returning an argument domain error if the user attempts to lock top-bit-set extents. – end note]

- The effects of placing an exclusive lock on top of a shared lock (lock upgrade) is implementation defined. It may be atomic or not. It may replace the shared lock, or not. It may silently not work as well.
- The effects of downgrading an exclusive lock to a shared lock of the same region is implementation defined. It may atomic, or not. It may release the exclusive lock, or not. It may silently not work as well.

[*Note:* Portable code should never perform lock upgrades or downgrades. Completely release the region, and take a fresh lock with the setting you want. Make sure to reexamine the region for changes by other code. – end note]

- Whether it is possible to unlock part of a region previously locked is implementation defined (some implementations may insist that what is unlocked is precisely what was previously locked i.e. they ‘stack’ region locks).
- Whether it is possible to unlock overlapping regions in a different order to the order of their locking is implementation defined.

[*Note:* Portable code should always unlock exact regions locked in a perfect reverse order, never taking locks on overlapping regions on the same inode, even if multiple handles are in use anywhere in the process including in third party code. – end note]

Throws: (i) Any failure returned by the kernel (ii) if a non-zero and non-infinite deadline is supplied and this handle does not support timed deadline i/o, may fail with an error comparing equal to `errc::function_not_supported` (iii) if a non-infinite deadline is supplied and that deadline expired, an error comparing equal to `errc::timed_out` will be thrown.

Complexity: Depending on implementation, this may be a latency preserving function with a fixed cost of execution. It may also be a latency degrading function of indeterminate execution time.

```
extent_guard try_lock(extent_type offset,
                    extent_type bytes,
                    bool exclusive = true) throws(file_io_error)
                    [[no_visible_side_effects]];
```

Effects: Identical in effects to writing:

```
1 lock(offset, bytes, exclusive, chrono::seconds(0));
```

6.15 Header `<io/map_handle>`

[[llfio.io.map_handle](#)]

Todo

6.16 Header `<io/mapped_file_handle>`

[[llfio.io.mapped_file_handle](#)]

Todo

6.17 Header `<io/mapped>`

[[llfio.io.mapped](#)]

6.17.1 Synopsis

[[llfio.io.mapped.synopsis](#)]

```
1 namespace std { namespace experimental { namespace io { inline namespace v1 {
2
3 template <class T> class mapped : protected span<T>
4 {
5 public:
6     ///! The extent type.
7     using extent_type = typename section_handle::extent_type;
8     ///! The size type.
9     using size_type = typename section_handle::size_type;
10    ///! The index type
11    using index_type = typename span<T>::index_type;
12    ///! The element type
13    using element_type = typename span<T>::element_type;
14    ///! The value type
15    using value_type = typename span<T>::value_type;
16    ///! The reference type
17    using reference = typename span<T>::reference;
18    ///! The pointer type
19    using pointer = typename span<T>::pointer;
```



```

20  ///! The const reference type
21  using const_reference = typename span<T>::const_reference;
22  ///! The const pointer type
23  using const_pointer = typename span<T>::const_pointer;
24  ///! The iterator type
25  using iterator = typename span<T>::iterator;
26  ///! The const iterator type
27  using const_iterator = typename span<T>::const_iterator;
28  ///! The reverse iterator type
29  using reverse_iterator = typename span<T>::reverse_iterator;
30  ///! The const reverse iterator type
31  using const_reverse_iterator = typename span<T>::const_reverse_iterator;
32  ///! The difference type
33  using difference_type = typename span<T>::difference_type;
34
35  public:
36  ///! Default constructor
37  constexpr mapped() noexcept {}
38
39  ///! Copies are disabled
40  mapped(const mapped &) = delete;
41  mapped &operator=(const mapped &) = delete;
42
43  ///! Moves are permitted
44  mapped(mapped &&) = relocates;
45  mapped &operator=(mapped &&) noexcept;
46
47  ///! Returns a reference to the internal section handle
48  const section_handle &section() const noexcept;
49  ///! Returns a reference to the internal map handle
50  const map_handle &map() const noexcept;
51  ///! Returns a span referring to this mapped region
52  span<T> as_span() const noexcept;
53
54  ///! Detach and destroy the map
55  ~mapped();
56
57  using span<T>::first;
58  using span<T>::last;
59  using span<T>::subspan;
60  using span<T>::size;
61  using span<T>::ssize;
62  using span<T>::size_bytes;
63  using span<T>::empty;
64  using span<T>::operator[];
65  using span<T>::operator();
66  using span<T>::at;
67  using span<T>::data;
68  using span<T>::begin;
69  using span<T>::end;
70  using span<T>::cbegin;
71  using span<T>::cend;
72  using span<T>::rbegin;
73  using span<T>::rend;
74  using span<T>::crbegin;
75  using span<T>::crend;

```

```

76     using span<T>::swap;
77
78     /*! Create a view of newly allocated unused memory, creating new memory if insufficient
79     unused memory is available. Note that the memory mapped by this call may contain non-zero
80     bits (recycled memory) unless 'zeroed' is true.
81     */
82     explicit mapped(size_type length, bool zeroed = false, section_handle::flag _flag = section_handle::
83         flag::readwrite) throws(file_io_error);
84
85     /*! Construct a mapped view of the given section handle.
86     explicit mapped(section_handle &sh, size_type length = (size_type) -1, extent_type byteoffset = 0,
87         section_handle::flag _flag = section_handle::flag::readwrite) throws(file_io_error);
88
89     /*! Construct a mapped view of the given file.
90     explicit mapped(file_handle &backing, size_type length = (size_type) -1, extent_type maximum_size =
91         0, extent_type byteoffset = 0, section_handle::flag _flag = section_handle::flag::readwrite)
92         throws(file_io_error);
93 };
94 } } } }

```

6.17.2 Class `mapped`

[[llfio.io.mapped.mapped](#)]

A convenience wrapper of `map_handle` and [[P1631](#)] `attached<T>` with owning semantics.

Remarks: The type `mapped` must meet the `MoveRelocating` concept.

6.17.2.1 Class `mapped` constructors

[[llfio.io.mapped.mapped.constructors](#)]

```
constexpr mapped() noexcept;
```

Effects: Constructs an invalid `mapped`.

```
explicit mapped(size_type length,
                bool zeroed = false,
                section_handle::flag _flag = section_handle::flag::readwrite)
    throws(file_io_error);
```

Effects: Creates owned new memory via constructing an unbacked internal `map_handle` with the given parameters.

Ensures: The attachment via [P1631](#) `in_place_attach<T>()` from the underlying newly created byte array of an array of `T`. Note that these will have indeterminate values unless `zeroed` is true, and may be invalid (i.e. uninitialised) object instances depending on the type of `T`.

Throws: Any failures of the construction of the internal `map_handle`.

```
explicit mapped(section_handle &sh,
                size_type length = (size_type) -1,
```

```

        extent_type byteoffset = 0,
        section_handle::flag _flag = section_handle::flag::readwrite)
throws(file_io_error);

```

Effects: Creates an owning view of the memory represented by the section handle via constructing an internal `map_handle` of length `length` with a byte offset into the section handle's memory of `byteoffset`. If `length` is all bits one, use the current length of the section handle divided by the size of `T`.

Ensures: The attachment via P1631 `in_place_attach<T>()` from the underlying byte array of an array of `T`.

Throws: Any failures of the construction of the internal `map_handle`.

```

explicit mapped(file_handle &backing,
                size_type length = (size_type) -1,
                extent_type maximum_size = 0,
                extent_type byteoffset = 0,
                section_handle::flag _flag = section_handle::flag::readwrite)
throws(file_io_error);

```

Effects: Creates an owning view of the storage represented by the file handle via constructing an internal `section_handle` to the file of length `maximum_size`, and an internal `map_handle` of length `length` with a byte offset into the section handle's memory of `byteoffset`. If `maximum_size` is zero, use the current maximum extent of the file for creating the section handle. If `length` is all bits one, use the current length of the section handle divided by the size of `T`.

Ensures: The attachment via P1631 `in_place_attach<T>()` from the underlying byte array of an array of `T`.

Throws: Any failures of the construction of the internal `section_handle` and `map_handle`.

6.17.2.2 Class `mapped` destructors

[`llfio.io.mapped.mapped.destructors`]

```
~mapped();
```

Effects: Unmaps any region mapped by a constructor.

Ensures: The detachment via P1631 `in_place_detach()` of the array of `T` into its underlying byte array before region unmap.

6.18 Header <io/native_handle>

[llfio.io.native_handle]

6.18.1 Synopsis

[llfio.io.native_handle.synopsis]

`native_handle_type` is already in the C++ 17 library as an implementation defined type, and appears in the Networking TS amongst other places.

The below does not propose replacing that implementation defined type, but it does suggest that a *disposition* be added to any implementation defined type if one is not already present. The disposition specifies metadata about the native handle, which greatly eases a number of interoperation issues such as trying to supply a process native handle type to a thread function, and so on.

```
1 namespace std { namespace experimental { namespace io { inline namespace v1 {
2
3 struct native_handle_type
4 {
5     /*! The type of handle.
6     bitfield disposition
7     {
8         invalid = 0U,    //!< Invalid handle
9
10        readable = 1U << 0U,    //!< Is readable
11        writable = 1U << 1U,    //!< Is writable
12        append_only = 1U << 2U, //!< Is append only
13
14        overlapped = 1U << 4U,  //!< Requires additional synchronisation
15        seekable = 1U << 5U,   //!< Is seekable
16        aligned_io = 1U << 6U,  //!< Requires sector aligned i/o (typically 512 or 4096)
17
18        file = 1U << 8U,        //!< Is a regular file
19        directory = 1U << 9U,   //!< Is a directory
20        symlink = 1U << 10U,   //!< Is a symlink
21        multiplexer = 1U << 11U, //!< Is a kqueue/epoll/iocp
22        process = 1U << 12U,   //!< Is a child process
23        section = 1U << 13U    //!< Is a memory section
24    }
25    disposition behaviour;    //!< The behaviour of the handle
26    union {
27        intptr_t _init{-1};
28        /*! A POSIX file descriptor
29        int fd; // NOLINT
30        /*! A POSIX process identifier
31        int pid; // NOLINT
32        /*! A Windows HANDLE
33        win::handle h; // NOLINT
34    };
35    /*! Constructs a default instance
36    constexpr native_handle_type();
37    ~native_handle_type() = default;
38    /*! Construct from a POSIX file descriptor
39    constexpr native_handle_type(disposition _behaviour, int _fd) noexcept;
40    /*! Construct from a Windows HANDLE
41    constexpr native_handle_type(disposition _behaviour, win::handle _h) noexcept;
42
43    /*! Copy construct
```

```

44 native_handle_type(const native_handle_type &) = default;
45 ///! Move construct
46 constexpr native_handle_type(native_handle_type &&o) noexcept;
47 ///! Copy assign
48 native_handle_type &operator=(const native_handle_type &) = default;
49 ///! Move assign
50 constexpr native_handle_type &operator=(native_handle_type &&o) noexcept;
51 ///! Swaps with another instance
52 void swap(native_handle_type &o) noexcept;
53
54 ///! True if valid
55 explicit constexpr operator bool() const noexcept;
56 ///! True if invalid
57 constexpr bool operator!() const noexcept;
58
59 ///! True if the handle is valid
60 constexpr bool is_valid() const noexcept;
61
62 ///! True if the handle is readable
63 constexpr bool is_readable() const noexcept;
64 ///! True if the handle is writable
65 constexpr bool is_writable() const noexcept;
66 ///! True if the handle is append only
67 constexpr bool is_append_only() const noexcept;
68
69 ///! True if overlapped
70 constexpr bool is_overlapped() const noexcept;
71 ///! True if seekable
72 constexpr bool is_seekable() const noexcept;
73 ///! True if requires aligned i/o
74 constexpr bool requires_aligned_io() const noexcept;
75
76 ///! True if a regular file or device
77 constexpr bool is_regular() const noexcept;
78 ///! True if a directory
79 constexpr bool is_directory() const noexcept;
80 ///! True if a symlink
81 constexpr bool is_symlink() const noexcept;
82 ///! True if a multiplexer like BSD kqueues, Linux epoll or Windows IOCP
83 constexpr bool is_multiplexer() const noexcept;
84 ///! True if a process
85 constexpr bool is_process() const noexcept;
86 ///! True if a memory section
87 constexpr bool is_section() const noexcept;
88 };
89
90 } } } }

```

6.18.2 Class `native_handle_type`

[`llfio.io.native_handle.native_handle_type`]

As `native_handle_type` is merely here for exposition and to remind readers of the same implementation defined type already in the C++ standard, it will not be explained further.

Remarks: The type `native_handle_type` must be `constexpr` constructible, and meet the `TriviallyCopyable` and `StandardLayout` concepts.

6.19 Header `<io/status_code>` [[llfio.io.status_code](#)]

6.19.1 Synopsis [[llfio.io.status_code.synopsis](#)]

Note that `status_code<>` is from [[P1028](#)] *SG14 status_code and standard error object for P0709 Zero overhead deterministic exceptions*.

```
1 namespace std { namespace experimental { namespace io { inline namespace v1 {
2
3 using file_io_error = errored_status_code<file_io_status_code_domain>;
4 // as-if:
5 {
6 public:
7 using errored_status_code<file_io_status_code_domain>::errored_status_code;
8
9 ///! Returns any first path associated with the errored status
10 filesystem::path path1() const;
11
12 ///! Returns any second path associated with the errored status
13 filesystem::path path2() const;
14 };
15
16 } } } }
```

6.19.2 Class `file_io_error` [[llfio.io.status_code.file_io_error](#)]

A status code capable of transporting a system code with information on how to fetch on demand up to two filesystem paths.

Remarks: The type `file_io_error` must be `constexpr` constructible, and meet the `MoveRelocating` concept. It is not permitted to allocate nor free memory during construction and destruction, however it may allocate memory upon an implicit conversion to `std::error`.

6.19.2.1 Class `file_io_error` constructors [[llfio.io.status_code.file_io_error.constructors](#)] [[llfio.io.status_code.file_io_error.constructors](#)] Todo

6.20 Header `<io/path_discovery>` [[llfio.io.path_discovery](#)]

Todo

6.21 Header `<io/path_handle>` [[llfio.io.path_handle](#)]

Todo

6.22 Header `<io/random_file_handle>` [llfio.io.random_file_handle]

Todo

6.23 Header `<io/section_handle>` [llfio.io.section_handle]

Todo

6.24 Header `<io/stat>` [llfio.io.stat]

Todo

6.25 Header `<io/statfs>` [llfio.io.statfs]

Todo

6.26 Header `<io/symlink_handle>` [llfio.io.symlink_handle]

Todo

7 Frequently asked questions

7.1 Why bother with a low level file i/o library when calling the kernel syscalls directly is perfectly fine?

1. This low level file i/o library defines *a common language* of basic operations across platforms. In other words, it chooses a common denominator across 99% of platforms out there. For example, if you append to a memory mapped file, that'll do the platform-specific magic on all supported platforms.
2. This low level file i/o library only consumes and produces trivially copyable, move relocatable and standard layout objects. Empirical testing has found that the optimiser will eliminate this low level library almost always, inlining the platform specific syscall directly. So, it is *no worse* in any way over calling the platform syscalls directly, except that this library API is portable.
3. Where trivial to do so, we encode domain specific knowledge about platform specific quirks. For example, `fsync()` on MacOS does not do a blocking write barrier, so our `barrier()` function calls the appropriate magic `fcntl()` on MacOS only where the `barrier()` is requested to block until completion.

4. This low level file i/o library is a bunch of primitives which can be readily combined together to build filesystem algorithms whose implementation code is much cleaner looking and easier to rationalise about than using syscalls directly.
5. We can provide deep integration with C++ language features in a way which platform specific syscalls cannot. Ranges, Coroutines and Generators are the obvious examples, but we also make a ton of use of `span<T>`, so all code which understands `span<T>` – or more likely the `std::begin()` and `std::end()` overloads it provides – automatically works with no extra boilerplate needed.

7.2 The filesystem has a reputation for being riddled with unpredictable semantics and behaviours. How can it be possible to usefully standardise anything in such a world?

That is a very good question. This proposal *passes through*, for the most part, whatever the platform syscalls do. If, for example, POSIX `read()` and `write()` implement the POSIX file i/o atomicity guarantees for file i/o as they are supposed to do, then:

1. A write syscall’s effects will either be wholly visible to concurrent reads, or not at all (i.e. no ‘torn writes’).
2. Reads of a file offset *acquire* that offset, writes to a file offset *release* that offset. Acquire and release have the same meaning as for atomic acquire and release, so they enforce a sequential ordering of visibility to concurrent users based on overlapping regions¹⁹.

These are very useful guarantees for implementing lock free filesystem algorithms, and are a major reason to use POSIX `read()` and `write()` (i.e. our proposed `file_handle`) instead memory maps (i.e. our proposed `mapped_file_handle`) because one can forego using any additional locking in the former. Major platform support for the POSIX read/write atomicity guarantees is pretty good in recent years²⁰:

	FreeBSD ZFS	Linux ext4	Win10 NTFS
Buffered i/o	Scatter-gather	No	Per buffer
Unbuffered i/o	Scatter-gather	Scatter-gather	Scatter-gather

Such domain specific knowledge requirements as this may seem to strongly recommend against standardisation into C++. However, I would counter with the point that lock free programming requires extensive domain specific knowledge. As does SIMD programming. And as does the standard library’s generic algorithms and containers themselves.

We in C++ have historically not shied away from requiring significant domain specific knowledge to fully realise the potential of the low level parts of the ecosystem. I don’t think we should exclude low level file i/o just because parts of it vary in semantics between operating systems. After all, even simple RAM varies in semantics between systems, yet we still manage to write C++ programs

¹⁹Many, if not most, filing systems actually implement a RW mutex per inode so their guarantees are rather stronger than POSIX requirements. One should not rely on this in portable code however!

²⁰Scatter-gather atomicity means that the entire of a scatter-gather buffer sequence is treated as an atomic unit. Per buffer atomicity means that atomicity is per scatter-gather buffer only.

which are reasonably performant and portable across a wide range of systems, by avoiding doing things which are not portable. The same applies to the filesystem, in my opinion.

7.3 Why do you consider race free filesystem so important as to impact performance for all code by default, when nobody else is making such claims?

Firstly, performance is only impacted if the host platform does not support direct syscall implementations for all the race free operations exposed by the proposed low level file i/o library, and the missing functionality must be emulated from user space. At least one major platform provides a full set (Microsoft Windows), and I have an enhancement ticket open for Linux²¹ to implement the missing support. If WG21 smiles upon this proposal, that should light a fire under kernels completing the missing syscall support.

I strongly take the opinion that correctness must precede performance, and as the filesystem is free to be concurrently permuted at any time by third parties, a correct implementation **requires** program code to be as impervious as possible to filesystem race conditions.

I appreciate that many do not share this opinion. A great many ran ext3 as their Linux filing system when it was demonstrably incorrect in a number of important behaviours²². Such users preferred maximum performance to losing data occasionally, and I don't mind any individual choosing that for their individual needs.

But international engineering standards must be more conservative. Choices made here affect everybody, including users where data loss must be avoided at all costs. Defaulting to race free filesystem is the safest choice. Without defaulting to race free filesystem, code written using this low level file i/o library would be much less secure, more prone to surprising behaviour, and end users of C++ code exposed to a higher risk of loss of their data.

8 Acknowledgements

Thanks to Nicol Bolas, Bengt Gustaffson, Chris Jefferson, Marshall Clow and Ben Craig for their feedback.

9 References

- [P0443] Jared Hoberock, Michael Garland, Chris Kohlhoff, Chris Mysen, Carter Edwards, Gordon Brown,
A Unified Executors Proposal for C++
<http://wg21.link/P0443>

²¹https://bugzilla.kernel.org/show_bug.cgi?id=93441

²²Feel fear after reading <http://danluu.com/file-consistency/>.

- [P0593] Richard Smith,
Implicit creation of objects for low-level object manipulation
<https://wg21.link/P0593>
- [P0709] Herb Sutter,
Zero-overhead deterministic exceptions
<https://wg21.link/P0709>
- [P0829] Ben Craig,
Freestanding proposal
<https://wg21.link/P0829>
- [P0939] B. Dawes, H. Hinnant, B. Stroustrup, D. Vandevoorde, M. Wong,
Direction for ISO C++
<http://wg21.link/P0939>
- [P1026] Douglas, Niall
A call for a Data Persistence (iostream v2) study group
<https://wg21.link/P1026>
- [P1028] Douglas, Niall
SG14 status_code and standard error object for P0709 Zero-overhead deterministic exceptions
<https://wg21.link/P1028>
- [P1029] Douglas, Niall
SG14 move = relocates
<https://wg21.link/P1029>
- [P1030] Douglas, Niall
Filesystem path views
<https://wg21.link/P1030>
- [P1095] Douglas, Niall
Zero overhead deterministic failure – A unified mechanism for C and C++
<https://wg21.link/P1095>
- [P1631] Douglas, Niall and Steagall, Bob
Object detachment and attachment
<https://wg21.link/P1631>
- [1] *Boost.Outcome*
Douglas, Niall and others
<https://ned14.github.io/outcome/>
- [2] Deukyeon Hwang and Wook-Hee Kim, UNIST; Youjip Won, Hanyang University; Beomseok Nam, UNIST
Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree
Proceedings of the 16th USENIX Conference on File and Storage Technologies (2018)
<https://www.usenix.org/system/files/conference/fast18/fast18-hwang.pdf>